UNIVERSITY OF CALIFORNIA
Santa Barbara

Memory Management for Multi-Language Multi-Runtime Systems on Multi-Core Architectures

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Michal Wegiel

March 2011

The Dissertation of
Michal Wegiel is approved:

Professor Amr El Abbadi

Professor Ben Zhao

Professor Chandra Krintz, Committee Chairperson

January 2011

Memory Management for Multi-Language Multi-Runtime Systems on Multi-Core Architectures

Dedication and Gratitude

I dedicate this dissertation to my family: my parents, Maria and Krzysztof, and my sister, Barbara, for their unconditional support and encouragement throughout all stages of my education.

I am deeply grateful to Chandra Krintz for all the support, guidance, mentorship, and help that she has provided during the entire process.

I would like to thank Ben Zhao, Amr El Abbadi, and Rich Wolski for serving on my Ph.D. committee.

I am grateful to Grzegorz Czajkowski and Laurent Daynes for being my mentors and collaborators during my internship at Sun Labs.

Finally, I would like to thank the staff, faculty, and fellow graduate students at the Computer Science department at UC Santa Barbara for their support and the opportunity to pursue this work.

Acknowledgements

The text of Chapters 3–7 is in part a reprint of the material as it appears in the conference proceedings listed below. The dissertation author was the primary researcher while the co-author listed on each publication directed and supervised the research which forms the basis for these chapters.

Chapter 3: Publication [159] in the ACM/SIGPLAN International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008).

Chapter 4: Publication [161] in the ACM/SIGPLAN International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2009).

Chapter 5: Publication [163] as UCSB Technical Report 2010-15.

Chapter 6: Publication [160] in the ACM/SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2008).

Chapter 7: Publication [164] in the ACM/SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010).

Curriculum Vitæ

Michal Wegiel

Education

2011	**Doctor of Philosophy in Computer Science,**
	University of California, Santa Barbara.
2006	**Master of Science in Computer Science,**
	University of Science and Technology, Krakow, Poland.

Experience

2006 – 2010	**Graduate Research Assistant,**
	University of California, Santa Barbara.
2004 – 2005	**Research Intern,**
	Sun Microsystems Laboratories, Menlo Park, CA.
2002	**Student Intern,**
	Motorola Global Software Group, Krakow, Poland.

Awards

2010	**Dissertation Year Fellowship,**
	University of California, Santa Barbara.
2006 – 2008	**Regents Central Fellowship,**
	University of California, Santa Barbara.

Publications

Michal Wegiel and Chandra Krintz: “Cross-Language, Type-Safe, and Transparent Object Sharing For Co-Located Managed Runtimes.” *In the ACM/SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2010.*

Michal Wegiel and Chandra Krintz: “Concurrent Collection as an Operating System Service for Cross-Runtime Cross-Language Memory Management.” *UCSB Technical Report #2010-15, 2010.*

Michal Wegiel and Chandra Krintz: “Dynamic Prediction of Collection Yield for Managed Runtimes.” *In the ACM/SIGPLAN International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2009.*

Michal Wegiel and Chandra Krintz: “The Single-Referent Collector: Optimizing Compaction for the Common Case.” *In the ACM/SIGPLAN Transactions on Architecture and Code Optimization (TACO), 2009.*

Michal Wegiel and Chandra Krintz: “XMem: Type-Safe, Transparent, Shared Memory for Cross-Runtime Communication and Coordination.” *In the ACM/SIGPLAN International Conference on Programming Language Design and Implementation (PLDI), 2008.*

Michal Wegiel and Chandra Krintz: “The Mapping Collector: Virtual Memory Support for Generational, Parallel, and Concurrent Compaction.” *In the ACM/SIGPLAN International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2008.*

Field of Study: Computer Science

Abstract

Memory Management for Multi-Language Multi-Runtime Systems on Multi-Core Architectures

Michal Wegiel

To manage the increasing complexity of software, developers employ a number of different strategies. These include using high-level, type-safe, object-oriented programming languages, executing applications within managed runtime environments (MREs), modularizing software into independent isolated components, and maximizing programmer productivity by implementing each component in the most-suitable language. Moreover, administrators and tools increasingly co-locate components on the same physical machine to better utilize multi-core systems via thread-level parallelism and to enable efficient cross-component communication. As a result, multi-language, multi-runtime systems that employ component co-location on multi-core shared-memory architectures are more and more common.

In such systems, memory management takes place within runtimes (intra-runtime) and between runtimes (cross-runtime). Intra-runtime memory management includes allocation and automatic reclamation of objects within an MRE. Cross-runtime memory management refers to communication, coordination, and object sharing across MREs. Both intra-runtime and cross-runtime memory management rely on the mechanisms and abstractions of the underlying operating system (OS) for efficient implementation.

The focus of our research is to identify ways to more effectively exploit extant OS functionality to improve intra-runtime and cross-runtime memory management in terms of performance as well as programming model. Specifically, we design, implement, and evaluate MRE extensions that leverage virtual memory, shared memory, and shared libraries to better coordinate memory management across the system layers.

For intra-runtime memory management, we develop new techniques to improve throughput, reduce pauses, increase yield, and enhance modularity of parallel and concurrent collectors. For cross-runtime memory management, we investigate type-safe, transparent object sharing between isolated MREs to enable cross-language communication and synchronization without expensive object serialization and explicit message passing.

Our empirical results indicate that our contributions significantly improve both intra-runtime and cross-runtime memory management by better leveraging OS support. We obtain large performance gains for parallel and concurrent collectors as well as inter-runtime communication over the state-of-the-art memory management systems. In addition, our techniques enhance the programming model for both application developers and runtime architects.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software applications and systems have become significantly complex as developers attempt to model and solve a wide range of scientific, engineering, and business problems as well as to provide automation, services, tools, and abstraction to users of diverse hardware platforms. Considering open-source software alone, there has been an exponential growth in the number of lines of code over the last decade [56] with the doubling time of around fourteen months. To manage the increasing software complexity, and thus make systems more reliable, easier to design, implement, deploy, maintain, and evolve, developers employ a number of approaches and methodologies, which include:

- **High-level programming languages.** Increasingly, programmers implement software using type-safe, object-oriented languages that target virtualized execution within managed runtime environments (MREs). Recent rankings for programming language popularity [153] show that unmanaged languages account for approximately 27% of software development activities today. The vast majority of newly-built systems employs managed languages, notably Java, PHP, Basic, C#, and Python.

Most MREs support automatic memory management (garbage collection) to simplify application development and improve system reliability. In addition, high-performance, scalable MREs typically provide incremental adaptive compilation, efficient threading and synchronization primitives, as well as dynamic extensibility via run-time class loading. Cross-platform portability, high-level abstractions, expressible programming model, and safe program execution, reduce the costs and effort required to develop and deploy large multi-layer software systems, such as enterprise applications, middleware, and web services.

- **Isolated components.** Developers commonly divide systems into multiple independent components that interact through well-defined interfaces and are otherwise isolated from one another. This software design pattern increases system modularity and reusability as well as simplifies the system architecture via abstraction. In addition, componentization makes software more reliable through fault isolation, separation of concerns, and encapsulation. Distinct components are typically run in separate MRE instances to take advantage of inter-process resource isolation and fault containment as well as to improve performance via more aggressive specializations in the MREs (e.g. choosing the best performing memory management strategy for a specific component).

Web frameworks, such as J2EE [93] and .NET [113], are example systems that partition applications into isolated components. They employ a three-tier architecture that consists of the presentation layer (the web container component), business logic (the application server component), and data source (the database component). Each tier is deployed using a separate MRE and cross-component interaction takes place via inter-process communication protocols.

- **Multi-language systems.** To increase programmer productivity and overall system performance, distinct components are often developed in different programming languages. For instance, large-scale distributed systems and applications at Facebook and Google are built from a wide range of backend services, each implemented using the language that is best suited to a particular purpose and functionality. Thrift [143] and Protocol Buffers [129] have been developed for efficient interoperation between such multi-language services. Another example is web applications: the presentation layer generates dynamic web pages using server-side scripting languages, such as dynamically-typed PHP and Ruby, while the database backend is typically implemented in general-purpose statically-typed languages such as Java, C#, and C/C++.

In addition, recent hardware architectures increasingly rely on providing greater numbers of processing cores instead of higher clock frequency in order to continue to deliver high performance [72]. Commodity computers today are equipped with processors that have four (desktops) to eight (servers) superscalar cores that share most of the memory hierarchy and commonly implement hyperthreading [55].

Because of the significant challenges with extracting parallelism from applications, developers and administrators attempt to fully utilize multi-core systems by co-locating multiple processes on a single machine. Systems that comprise multiple isolated components simplify and facilitate such configuration and placement since each component is independent of others and the communication protocols operate in the same way regardless of whether interoperating components are co-located or distributed. Co-location also benefits application performance by reducing the cost of cross-component communication.

As a result of these software and hardware trends as well as widely-used development and deployment strategies, it is increasingly common for software designers to architect systems that are *multi-language* (i.e. components are implemented in different languages) and *multi-runtime* (i.e. each component is run in a separate MRE). In addition, multiple isolated components are more and more often *co-located* on *multi-core shared-memory architectures* and use inter-process communication mechanisms for interaction and coordination.

In such systems, memory management takes place at two levels: within runtimes (*intra-runtime*) and between runtimes (*cross-runtime*). Figure 1.1 depicts this schematically. Multiple managed runtimes, potentially for different programing languages (Java and Python in this case), are co-located on a single multi-core shared-memory machine. Intra-runtime memory management includes object allocation and automatic reclamation of unreachable objects (garbage collection), in MRE-private heaps. Cross-runtime memory management refers to cross-MRE communication by object sharing (i.e. via direct pointers to objects in a shared heap) and by message passing (i.e. via sending serialized objects through channels).

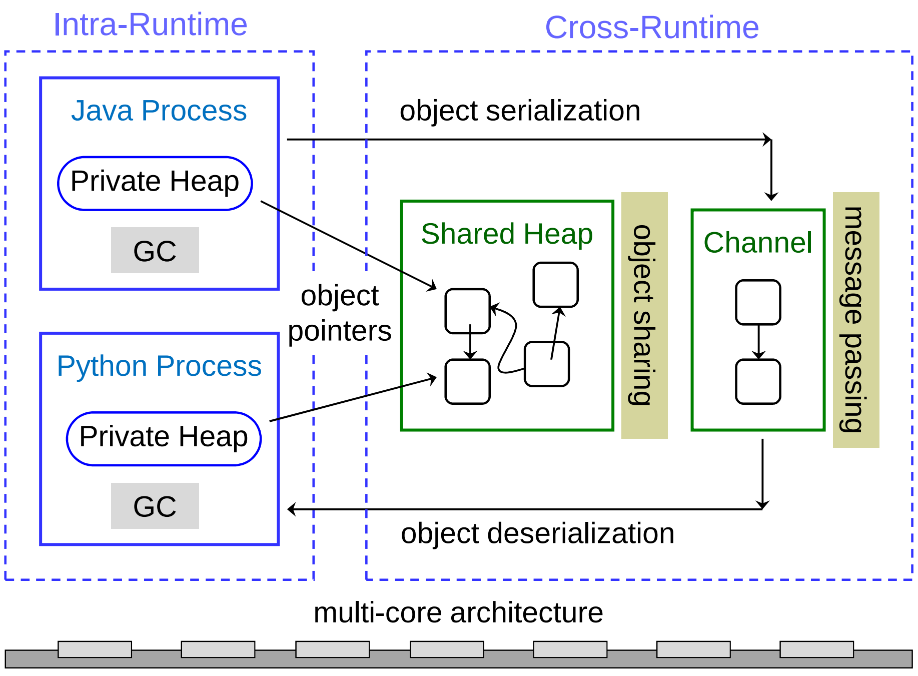


Figure 1.1: Intra-runtime and cross-runtime memory management in multi-runtime multi-language systems deployed on multi-core shared-memory architectures.

Both intra- and cross-runtime memory management impacts system *performance*. The runtime cost of garbage collection (GC) is between 20% to 70% of execution time, depending on the heap size, application behavior, and the GC algorithm [159, 161]. Similarly, the overhead of cross-MRE interaction via remote procedure calls and messages in request-intensive on-line transaction processing systems constitutes a large portion of the end-to-end performance [160, 164].

Memory management is a subsystem that significantly affects the *programming model*, both at the application and system level. A large percentage of the programming effort required to implement an application or an MRE is typically devoted to memory management. Thus, providing the right primitives and mechanisms for intra- and cross-runtime memory management is key to improving the programming model for application and MRE architects.

Memory management at both levels relies on the mechanisms and abstractions provided by the underlying *operating system* (OS). MREs need to fully leverage OS support in order to implement resource-efficient memory management mechanisms that constitute an expressible and flexible programming model.

1.1 Thesis Question

The primary research question that we explore in this dissertation can be stated as follows:

> *How can we improve intra-runtime and cross-runtime memory management in multi-language, multi-runtime systems that co-locate multiple software components on multi-core shared-memory architectures, by taking advantage of operating system support?*

To answer this question, we design, implement, and evaluate *MRE extensions* in order to better coordinate memory management across the system layers. We consider MREs for both statically-typed and dynamically-typed languages. Our goal is to improve system *performance* as well as the *programming model* for both application and MRE developers. We investigate OS support for *virtual memory*, *shared memory*, and *shared libraries*.

Table 1.1 summarizes the *two-dimensional design space* that we cover with this dissertation. Rows represent the two metrics that we use (performance and programing model) and columns correspond to the two levels of memory management that we investigate (intra- and cross-runtime). For each metric-level pair we report the OS mechanism(s) that we leverage to improve memory management.

A primary goal of ours is to improve performance. For intra-runtime memory management, we aim at designing new parallel and concurrent collectors in order to reduce GC pauses and execution time overhead as well as increase collection yield. For

Metric	Intra-Runtime	Cross-Runtime
Performance	virtual memory (3, 4) shared libraries (5)	shared memory (6, 7)
Programming Model	shared libraries (5)	shared memory (6, 7)

Table 1.1: Two-dimensional design space in memory management that we investigate. Rows are metrics and columns are the two levels of memory management in multi-runtime systems. For each point in the design space, being a metric-level pair, we list the OS mechanism that we leverage to improve memory management. In parentheses, we show the numbers of chapters that describe the corresponding systems that we contribute.

cross-runtime memory management, we investigate the design of type-safe, transparent object sharing to increase throughput and decrease latency of cross-runtime communication by avoiding object serialization.

Another key goal of our research is to improve the programming model. For intra-runtime memory management, we aim at enhancing the modularity of MRE GC implementation, by investigating the design of a portable GC library that decouples GC from MRE internals. For cross-runtime memory management, we aim at designing a new type-safe cross-MRE communication primitives based on shared memory that are simpler and more transparent than explicit message passing.

1.2 Dissertation Organization

We organize the remainder of this dissertation as follows. We first provide background information, discuss terminology, state-of-the-art systems, open problems, and limitations in intra-runtime and cross-runtime memory management, in Chapter 2.

Chapters 3–7 describe the five systems that we contribute to address our thesis question and that represent separate points in the design space shown in Table 1.1. In this table, the parenthesized values are corresponding chapter numbers. Each of these five chapters starts with motivation and problem statement, then discusses the design and implementation details, followed by experimental evaluation, related work, and conclusions. Chapters 3–5 focus on intra-runtime memory management while Chapters 6 and 7 target cross-runtime memory management. OS support for virtual memory is discussed in Chapters 3 and 4, for shared libraries in Chapter 5, and for shared memory in Chapters 6 and 7. Techniques for improving performance are described in Chapters 3–7 while enhancing the programming model is the subject of Chapters 5–7. Chapter 8 summarizes our contributions and discusses future research directions.

Chapter 2

Background

In this chapter, we provide background on and survey state-of-the-art in intra-runtime and cross-runtime memory management techniques. Of particular interest to us are systems deployed on multi-core shared-memory architectures and ones using operating system support for memory management. We overview recent advances in garbage collection and cross-runtime communication and sharing as well as discuss limitations of extant systems.

2.1 Intra-Runtime Memory Management

In this section, we discuss automatic memory management techniques employed by type-safe programming language runtimes. We overview key concepts and terminology related to state-of-the-art garbage collectors, recent system and algorithmic advances in this area, and limitations of extant approaches. We also provide background on OS-assisted collectors.

2.1.1 State-of-the-Art GC Techniques

Managed runtime environments (MREs) for portable, object-oriented, type-safe programming languages, both statically-typed (e.g. Java and C#) and dynamically-typed (e.g. Python and Ruby) provide garbage collection (GC) to support memory safety and simplify programs by automating memory management. While increasing programmer productivity and application reliability, GC can negatively impact both application throughput (through additional processing) and interactivity (through pauses). Minimizing the costs of GC to match or exceed those of explicit memory management has been the subject of active research for several decades [97]. An excellent introduction into and overview of the GC literature can be found in [99, 165, 166, 167]. Modern GC techniques include parallel, concurrent, and on-the-fly GCs that exploit multi-core architectures to reduce or eliminate pauses and scale multi-threaded applications while maintaining high throughput, and that use OS support to better coordinate runtime and kernel memory management. High-performance MREs typically employ generational GCs, which outperform other GC schemes in the common case.

Terminology

General-purpose MREs predominately use *tracing* GCs, which work by first determining which objects are reachable (live) from *roots* and then reclaim the memory occupied by the remaining (dead) objects. In tracing GCs, the first phase is usually called *marking* and amounts to computing the transitive closure of the objects reachable from the roots. The second phase can either be *sweeping* [166, 180, 59], where live objects are not moved and dead blocks are added to free lists, or *compaction* [101, 159, 1], where live objects are slided to form a contiguous memory region.

Reference counting [172, 166, 171], being the other class of GC algorithms, is an incremental scheme where each object maintains a counter for incoming references which is updated on every pointer store. Dead objects are garbage collected when they become unreferenced. Due to such limitations as storage overhead, problems with detecting dead cycles, and poor efficiency, reference counting is not as commonly used as tracing (one notable example is Python). Some collectors combine tracing and reference counting [30].

In strongly-typed languages (like Java), GC is *precise*, i.e. it can accurately identify all object pointers inside the thread stacks and in the C heap. In languages providing weaker type-safety (like C++), GC is *conservative* [22, 23, 31, 35, 181], i.e. all memory locations containing a value that resembles a valid pointer are treated as pointers. Conservative collectors can suffer from memory leaks due to misinterpretation of regular data as pointers.

Serial GCs [165] employ a single thread to perform collection. Modern MREs use *parallel* GCs [34, 67, 4] to scale on SMPs and multi-core platforms. Parallel marking typically employs static partitioning for the root set and a dynamic load balancing scheme, such as work stealing [67], for object graph tracing.

Application threads (*mutators*) are suspended for collection by *stop-the-world (STW)* GCs [99, 165, 101, 159]. In contrast, *concurrent* GCs [20, 19, 34, 82, 122] perform most work in the background, without stopping the application. Compared to STW GCs, concurrent collectors impose much shorter pauses, however, they require resource over-provisioning in terms of the number of processing cores and memory footprint. Many concurrent GCs require *all* mutator threads to be halted briefly (for example, at the start and end of each GC phase), others stop mutator threads one at a time (*on-the-fly* GCs [61, 62]). *Incremental* [140, 17, 165] collectors interleave small bits of collection work with the mutator activity such as allocation and pointer stores. In *real-time* collectors pause times imposed by GC are bounded within any given time interval, which is often specified as the minimum mutator utilization (MMU).

Compacting GCs [47, 99, 100, 67, 101, 46, 85, 1] eliminate fragmentation in the heap by consolidating live objects into a single contiguous region in memory. *Copying* GCs [166, 144, 87, 58, 14] divide the heap in two semi-spaces and in every GC cycle evacuate live objects from the currently-used semi-space to the other. This approach is most efficient when the percentage of live objects is small on average, i.e. when objects have short lifetimes.

Generational GCs [16, 87, 155, 133, 168] group objects into separate sub-spaces called generations based on the object age. Typically, there is a young generation and an old generation. New objects are allocated in the new generation and are later promoted to the old generation if they survive a threshold number of collection cycles. Two properties are key for the effectiveness of generational GCs. First, most objects die in the young generation (weak generational hypothesis [155]). Second, there are relatively few references from the old generation to the young generation.

Generational GCs focus collection efforts on the young generation which is expected to contain mostly dead objects and this way GC yield is maximized. To enable independent collection of the young generation, a *write barrier* [84, 180, 125] is used to keep track of pointers from the old generation to the young generation. Generational GCs have established themselves as a de-facto standard for any high-performance general-purpose MRE, as they tend to provide superior throughput in practice. In a generational heap layout, the young generation typically uses a copying collector, to exploit the weak generational hypothesis. In older generations, where most data is expected to be live, MREs usually employ either a sweeping or a compacting collector. Compacting GCs eliminate fragmentation (thereby reducing space overhead) and enable simpler and more efficient linear (bump-pointer) allocation. Sweeping GCs trade fast non-moving collection for slower allocation.

Real-time [4, 15, 12] GCs guarantee fully predictable collection behavior for tasks with bounded allocation rate. Pause times are bounded and carefully scheduled to guarantee the desired MMU in a deterministic and consistent manner. Real-time GCs are considered special-purpose, targeted at embedded systems with hard or soft real time requirements. In general-purpose MREs, concurrent GC is usually sufficient as it can provide short pause times in the common case.

Performance Metrics

Two primary measures of garbage collection performance are *throughput* and *pauses* [99, 166]. Throughput is the percentage of total time not spent in garbage collection, considered over long periods of time. Throughput includes time spent in allocation. A common approach to comparing throughput of different GCs is measuring the application execution time. Pauses are the times when an application appears unresponsive because garbage collection is occurring. Most GC algorithms, including concurrent ones, need to halt program threads at least once per collection cycle to avoid interference with mutators.

Other important metrics are *footprint* and *promptness* [85]. Footprint is the working set of a process, measured in pages and cache lines. On systems with limited physical memory or many processes, footprint may dictate scalability. Promptness is the time between when an object becomes dead and when the memory becomes available, an important consideration for distributed systems, including remote method invocation (RMI). Promptness may be affected by finalization, which typically defers object collection until the next GC cycle.

A commonly-employed metric for the evaluation of collector-imposed pauses are minimal mutator utilization (MMU) curves [43] that lend insight into the distribution of GC pauses across program execution. Mutator utilization for a given time window w is defined as the fraction of the time that the mutator (as opposed to the collector) executes within the window w. Minimum mutator utilization for a window of a specific size s is the lowest mutator utilization for all time windows of size s across the program execution. Thus, the x-intercept of a MMU curve is the maximum pause time and the asymptotic y-value corresponds to the application throughput. MMU curves are especially useful for evaluating concurrent and real-time GCs.

For parallel GC, an important metric is scalability [159, 1], which is typically expressed as speedup. To measure speedup, GC is run with 1 to p parallel threads for a fixed workload (unscaled speedup) or for an increasing workload (scaled speedup). GCs that employ load balancing, e.g. work stealing, often achieve nearly-linear speedup.

2.1.2 OS-Assisted GC

Garbage collectors that coordinate memory management with the OS kernel typically do so by taking advantage of the available virtual memory operations. Appel and Li [5] describe a number of ways that user-land programs can use to exploit the virtual memory subsystem, including user-level signal handlers, multiple mapping, and page protection. A similar study that includes performance evaluation as well as recommendations for exposing dirty-bit information to MREs can be found in [83]. While some of these techniques require OS modifications, most are portable and standardized by the Portable Operating System Interface (POSIX). A good introduction to memory management in modern OSes can be found in [150, 149, 142, 114].

Virtual Memory

General-purpose operating systems support virtual memory to isolate address spaces of distinct processes, abstract away such characteristics of physical memory as limited size and non-contiguity, and provide a convenient uniform linear address space to programs. Most implementations divide the virtual address space of a process into *pages*, whose size is typically 4KB. Virtual addresses used by programs are converted to physical addresses used on the system bus. This process is called *memory mapping* or *address translation* and relies on hardware support for efficiency. The mapping between virtual pages and physical page frames is stored in a data structure called *page* *table*. OS kernel is responsible for creating and maintaining pages tables but employs the CPU memory management unit (MMU) to translate addresses. To accelerate address translation CPUs use a small associative memory, called the translation lookaside buffer (TLB), to cache mappings for recently accessed virtual pages. On each virtual memory access the CPU performs a TLB lookup. If a TLB entry is found, CPU can calculate the physical address immediately. On a TLB miss, a page table walk is performed. If the needed mapping is found in the page table, MMU inserts a new entry into the TLB. Otherwise, a TLB miss fault is generated. The fault is intercepted by the OS, which invokes the TLB miss handler. TLBs, being implemented in hardware, usually use simple replacement policies, like not recently used (NRU). The OS occasionally invalidates the entries in TLB, for example when context switching or swapping a page in or out.

Most operating systems use *demand paging* to avoid setting up mapping for pages that are never accessed (most processes allocate much more virtual memory than they ever use at any given point in time). With this approach, the virtual address space starts out empty and all virtual pages are marked as *not present*. When a page is accessed for the first time, the MMU generates a *minor page fault*. The kernel handles minor faults by allocating a new page frame, zeroing or loading its content, and updating the page table by inserting a new entry.

On memory pressure, the kernel uses *swap space* to evict pages that are unlikely to be accessed in the future. Most systems use some approximation of the least recently used (LRU) policy that has been shown to often perform almost as well as the provably optimal replacement policy. To implement LRU, every memory access would have to update a data structure, a solution too expensive to be practical. Commonly-used approximations are *clock replacement* and NRU. In addition to the page replacement policy, the kernel uses a memory balancing policy, which determines how much memory the OS can use for kernel buffers (e.g. the page cache) and how much to devote to backing virtual pages. To implement swapping, pages tables reserve two bits per page, indicating whether a specific page is *dirty* and has been recently *accessed*. These bits are set by hardware upon memory store (dirty bits) and memory reference (accessed bits). The kernel preferentially swaps out pages with both bits cleared.

Each page is associated with a set of permission bits that control if the content of a particular page can be read, written, and executed. Upon an unpermitted access, a page protection violation is raised and the process receives the SEGV signal. Permission bits enable safe sharing of page frames. Modern OSes implement the *copy-on-write* policy, where unmodified pages can be shared across processes – this commonly pertains to shared libraries and facilitates efficient process cloning.

The address space of a process consists of user space and kernel space. The latter is only accessible in the privileged mode. The kernel space is typically identity mapped, i.e. physical addresses can be computed by adding/subtracting a constant offset to virtual addresses. The kernel provides two allocators for physical memory: a *buddy allocator* that allocates physically contiguous and properly-aligned memory which is always some power-of-two (allocation order) times the page size and a *slab allocator* that better supports smaller allocations via kernel memory caches, each managing objects of a particular fixed size. The kernel also supports virtual memory allocations.

Pages tables in Linux/x86 are kept in physical memory in the identity-mapped kernel segment. As such, page tables are never swapped out, and there is no nested major page faults. Each page table is a multi-way tree, which logically has three levels: *global directory*, *middle directories*, and *PTE directories*. Each directory occupies one page frame and has a fixed size. Entries in the global and middle directories are either not present or they point to a directory in the next level of the tree. The leaves comprise page table entries (PTEs). Address translation works by splitting a virtual address into three indices (pgd, pmd, and `pte`, which correspond to the global, middle, and PTE directories) and a page offset. The kernel starts at the root (the global directory) and retrieves an entry stored at index pgd. This entry points to a middle directory. Similar process is repeated for the remaining two levels using pmd and pte. Having obtained the PTE, the system can calculate the physical address by combining the page frame address with the offset. Each PTE contains a page frame number (PFN), page permissions (read, write, and execute) as well as three bits determining if a page is present, dirty, and accessed. Page tables are loaded by copying the pointer to the global directory into the cr3 (page table base) register (the side-effect is TLB flushing). Linux organizes pages into three lists: active, inactive, and evicted. Active pages are managed using the CLOCK algorithm. Inactive pages are protected so that their access triggers a minor fault and moving into an active list. Access to evicted pages triggers a major fault. The active list is bounded in size, and implements a FIFO discipline when moving entries to the inactive list.

The IA-32 architecture supports memory segmentation in addition to paging. A segment is a contiguous region of virtual memory and a set of attributes that define access rights. *Global descriptor table (GDT)* and *local descriptor table (LDT)* keep tract of the segments in use. The system starts with a *logical address* that consists of a *segment selector* and an offset. The segment selector is an index into a descriptor table (GDT or LDT). Each memory access is checked against the segment boundaries and a fault is raised if the check fails. A 32-bit *linear address* is computed by adding the offset value to the segment base address. At this point the system moves on to paged address translation (TLB/page tables) to obtain the physical address. Segmentation enables many linear address spaces and aids sharing and protection. Unlike paging, it is visible to the user programs (compilers define their own semantics).

The POSIX Interface

OSes that are fully or partially compliant with POSIX, provide a standard interface allowing to manipulate virtual memory mapping or to impact the strategy the kernel uses to manage memory for a specific process. For example, Linux provides a number of system calls (kernel API) that enable to influence memory mapping (`mmap, munmap`, and `mremap`), page protection (`mprotect`), page pinning (`mlock` and `munlock`), and page swapping (`mincore, msync`, and `madvice`). Shared memory segments are allocated using shmget and controlled via shmctl. Processes attach/detach to/from shared segments via `shmat/shmdt`. Double mapping, where two different virtual addresses map to the same physical address, a technique often used in GC, is implemented by double attachment to a single shared segment from a process.

GCs leveraging OS support, interact with the kernel via system calls (downcalls) and signals (upcalls). For example, concurrent compactors protect virtual pages that are yet to be processed, intercept the SEGV signal, and process pages incrementally as mutators access them and trigger faults. GCs that manipulate virtual memory mapping, do so by invoking mmap/munmap system calls. Frequent interaction with the kernel imposes a certain overhead associated with kernel entry/exit. The kernel entry path includes switching to the kernel stack, saving scratch registers, and invoking an interrupt handler (system calls are software interrupts). The kernel exit path comprises a potential task scheduler invocation, signal delivery, restoration of scratch registers, and switching to the user stack. All these actions introduce latency. However, MREs can significantly benefit from well-architected OS support.

OS-GC Interaction

Extant garbage collectors interact with the OS in a wide array of contexts: to support concurrent marking and compaction [101, 46], to avoid GC-induced paging [173, 174, 79, 179, 77, 80, 175], to reduce space overhead by dynamic mapping/unmapping [136], to predict application working set size [175, 77], to optimize GC triggering and improve GC yield [39, 173, 175], and to provide GC as a system/language library [35, 34]. Several of these systems are worth discussing in more detail.

The Compressor [101] relies on virtual memory operations to enable concurrent compaction. The GC uses page protection to capture accesses to objects that have not yet been relocated. Such accesses trigger traps. The trap handler takes care of incremental (page by page) compaction. To allow the GC to access protected pages while trapping the application, the Compressor uses double mapping: a single physical page is mapped twice in the virtual address space (with and without access protection). In addition, the Compressor dynamically unmaps pages whose content has been copied to avoid the space overhead of a copying collector.

The Pauseless GC [46] uses similar mechanisms as the Compressor (dynamic page (un)mapping and page protection), however it relies on custom hardware and OS (both optimized to run Java) to implement efficient, scalable, pauseless compaction. The GC employs hardware read barriers that execute in one cycle and invoke user-mode traps once a stale reference is encountered. The Pauseless GC uses an additional intermediate privilege level (between the user and kernel mode) for fast execution of traps. In this mode, protected pages can be modified without the overhead of the OS entry/exit path. The GC implements safepoints on top of fast cooperative preemption via interrupts. Safepoint-checking instructions (e.g. back branches) check for a pending per-cpu safepoint interrupt and raise user-mode traps.

The Bookmarking GC [80] cooperates with the kernel memory management subsystem to avoid paging in cases where there is not enough physical memory in the system. The GC computes conservative summaries of object graph connectivity for the evicted pages (to avoid accessing swapped-out pages during GC and thus prevent frequent major page faults). The OS kernel sends a signal whenever it is about to evict a page or an evicted page has just been loaded back. The GC responds to the eviction signal by trying to find an empty page and if that is not possible by bookmarking the page it has selected for eviction.

CRAMM [175] dynamically resizes the heap to maximize application throughput and minimize paging by using statistical page reference information collected in the OS kernel. CRAMM estimates the working set size (WSS) for a process based on an LRU reference histogram. The system computes the desired heap size at runtime to keep the overall performance cost of swapping below a specific threshold (5% of application execution time). CRAMM extends the OS kernel by modifying the active/inactive lists implementation and leverages the CLOCK algorithm to compute detailed per-process LRU reference histograms. OS-GC interaction takes place through system calls after each GC, when GC requests WSS estimation from the kernel and resizes the heap accordingly.

The goal of MicroPhase [173] is to increase GC productivity (yield) by triggering GC at points when many objects become unreachable. The system recognizes program phases and proactively invokes GC during phase transitions. This approach strives to exploit the observation that allocation pauses (characterized by low allocation rates) typically correspond to points when GC is productive. MicroPhase cooperates with the OS kernel to implement efficient detection of phase transitions. The key challenge is the need to distinguish allocation pauses from execution pauses (e.g. thread blocking on I/O operations). The system extends the OS kernel by introducing per-thread CPU-cycle counters that are incremented based on hardware performance counters whenever a thread is scheduled for execution. Total CPU-cyles for a given thread can be read via a system call.

Boehm GC [35] is a conservative mark-sweep garbage collector for C/C++ (or a memory leak detector) distributed as a C library, portable across most operating systems. The GC has a simple interface, essentially consisting of allocation functions modeled after malloc/realloc in the standard C library. Boehm GC has been used in a variety of language runtimes and is available as a prepackaged library in several major Linux distributions.

2.1.3 Limitations

Recent advances in computer architecture have invalidated many of the assumptions that were the basis of the design and implementation of garbage collectors in the past, while opening the opportunities for new research directions. Hardware trends, such as increasing reliance on thread-level parallelism rather than on clock frequency to improve performance, the growing memory-processor performance gap, bus and cache contention on multi-core systems, and large 64-bit address spaces invite revisiting GC design. At the same time GC has become a crucial component of state-of-the-art MREs, which are the runtime platform for a wide range of software today, especially server-side middleware, web, and application servers.

The design and implementation of state-of-the-art general-purpose GCs is increasingly centered around scalability, low pauses, and high throughput. As a result, parallel, concurrent, and on-the-fly GCs receive more and more attention. In addition, the interaction between the collector and virtual memory has recently gained significant interest [101, 175, 80, 46]. Virtual-memory-oblivious GCs, while being the focal point of most prior work, can no longer provide the required level of efficiency on new hardware and software platforms. The design of modern GC must take into account the following aspects:

- **Parallelism.** GC needs to scale on both multi-processor and multi-core architectures. Collectors should employ parallel threads and use dynamic load balancing to achieve close-to-linear speedups.

- **Concurrency.** To guarantee a high level of interactivity, required for GUI applications as well as server-side software that needs to handle client requests at very high rates and with low latency, GC cannot impose lengthy pauses. Modern machines are over-provisioned (in terms of both memory and processing cores), which makes them well-suited for concurrent GC.

- **Large address space.** Modern 64-bit platforms provide applications with large virtual memory, which creates an opportunity for GCs to perform memory management by aggressive (re)mapping operations. For example, the virtual space occupied by unreachable objects does not have to be filled by live objects but can be instead remapped to a new area in the address space. This way, GC can leverage the level of indirection associated with address translation.

- **Memory hierarchy.** The widening memory-processor performance gap renders object copying and pointer updates increasingly expensive. Modern GCs should avoid such operations by adhering to non-moving collection as much as possible.

- **Abundant physical memory.** Space overhead is no longer a major consideration as long as it can be bounded (predictable) and does not exceed a reasonable percentage of the heap size. In the past, high memory footprint used to be a disqualifying property for a GC. Today's MREs can afford trading space overhead for performance.

- **Design complexity.** State-of-the-art systems consist of many layers both in software and in hardware and, in consequence, their performance characteristics are difficult to analyze and predict accurately. Increasingly, system architects prefer a minor performance penalty to an overly complex solution. Simplicity is an important design goal in modern GC algorithms.

- **Cross-layer coordination.** MREs should arrange for cooperative interaction with the OS memory manager. For instance, the GC memory access pattern should not conflict with the page replacement policy used by the OS kernel. In addition, MREs should use the available OS facilities in a more effective way.

- **Principle of locality.** It is important to preserve the order or objects, as it was created by an application, as this order reflects temporal/spatial locality of access.

- **Object clustering.** Empirical analysis of modern benchmarks indicates that objects exhibiting similar life spans tend to be spatially clustered in the heap. This statistical observation enables the GC design where reclamation granularity is bigger than a single object (for example page-based GC).

While recent GC algorithms [101, 46, 57, 59, 80, 175] have been increasingly addressing the above design aspects, there still remain many limitations to overcome. The most important ones that affect GC performance and interactivity are:

- **Object moving and pointer adjustment.** State-of-the-art GCs that perform parallel/concurrent compaction, move objects in the heap. This is expensive because it causes significant cache/memory traffic. In addition, moving objects makes the GC design more complex by introducing additional phases (passes over the heap).

- **Unproductive GCs.** Extant GCs that optimize GC triggering to increase GC yield are either based on heuristics related to program behavior or rely on offline profiling. These systems are unable to eliminate unproductive GCs in a generic way with low overhead and good accuracy.

- **Unexploited clustering.** Although dead object clustering is a widely-observed phenomenon (objects with similar lifetimes tend to be co-located in the heap [167]), to date, GC systems have taken little advantage of this statistical property. Now that 64-bit address space is increasingly commonplace, there are much more opportunities to exploit object clustering in the heap.

- **OS-oblivious GC.** Most GCs published in the literature do not leverage OS support for virtual memory and those that do [101, 46] do not eliminate object moving and low-yield collections. Existing systems do not tap the full potential of OS-assisted cross-layer memory management.

- **Complex, monolithic GC.** GC is one of the most complex subsystems in modern runtimes and for performance reasons is often tightly-coupled with other subsystems such as a dynamic compiler. Extant efforts aimed at modularizing GC either do not target modern, concurrent GC [35], or have limited portability and high overhead because are part of memory management frameworks written for and in high-level languages [28, 29].

The GC systems described in Chapters 3, 4, and 5 address these limitations by leveraging OS support for virtual memory and system libraries.

2.2 Cross-Runtime Memory Management

In this section, we overview inter-process communication mechanisms, both in the context of an operating system and a language runtime. We provide background on key concepts and terminology related to message passing and shared memory. In addition, we discuss state-of-the-art techniques for cross-runtime communication, such as remote procedure calls and object sharing, and point out their limitations.

2.2.1 State-of-the-art Inter-Process Communication

Inter-process communication (IPC) enables the exchange of data among multiple processes potentially running on multiple computers connected by a network. General-purpose operating systems, e.g. UNIX, typically provide several IPC mechanisms [9, 147, 105], such as message passing, synchronization, shared memory, and remote procedure calls (RPC). An efficient interprocess communication facility enables system decomposition across address space boundaries, which promotes failure isolation, extensibility, and modularity.

Signals and pipes are the oldest OS-level IPC mechanisms. With System V, AT&T introduced three additional forms of IPC: message queues, semaphores, and shared memory. POSIX equivalents of System V mechanisms define a slightly different API but offer similar functionality.

Signals are asynchronous events that can be generated by processes (provided that they have the correct privileges) or the OS kernel (for instance in response to a keyboard interrupt or an error condition such as when a process attempts to access a non-existent location in its virtual memory). There is a predefined set of signals in the system. Processes can either ignore or intercept most of the signals, with the exception of STOP (which causes a process to halt its execution) and KILL (which terminated a process). Ignored signals are handled by the kernel which performs the default actions required for them, e.g. for the floating point exception the kernel saves a core dump and terminates the process. Signals have no relative priorities and ordering. There is no mechanism for handling multiple signals of the same kind (e.g. a process cannot tell how many continue (CONT) signals it received). Signals are delivered on the kernel exit path only and therefore there may be some delay between generating a signal and presenting it to a process. Processes blocked in uninterruptible system calls are not awoken by signals. Calling a signal handler is processor-specific as it requires switching between the kernel and user mode. This is typically implemented by manipulating the stack and registers of the process. The program counter is set to the address of its signal handling routine and the parameters to the routine are added to the call frame or passed in registers (depending on the calling convention used). POSIX systems allow a process to block other signals while a particular signal handling routine executes.

Pipes are unidirectional byte streams which connect the standard output from one process into the standard input of another process. Neither process is aware of this redirection and behaves just as it would normally. It is common for the shell to set up temporary pipes between processes. As a process writes to a pipe, bytes are copied into the shared data page, from which they are later copied back as another process reads from the pipe. Pipes are synchronized by the kernel for exclusive read/write access. Reading or writing may block if there is nothing to read or no more room to store bytes. Some OSes, like Linux, support named pipes (also known as FIFOs) which are permanent entities in the file system.

The System V IPC mechanisms share common identification and authentication methods. Each IPC object has a unique identifier, generated based on a key that must be mutually agreed upon by processes that use a particular object. Access to IPC objects is granted based on a set of permissions.

A *message queue* is an internal linked list within the kernel address space. Multiple readers and writers can use a single message queue simultaneously. Message queues provide asynchronous communication (the sender and the receiver do not need to use the queue at the same time). Each time a process attempts to write a message to a queue and there is enough room in the queue, the message is copied from the user space to the kernel space and enqueued using the FIFO discipline. The kernel blocks the process if the queue is full. Each message is tagged with an application-specific type, agreed upon by the communicating processes. A reading process may specify the message type when retrieving a message from the queue. If no messages match the given type or the queue is empty, the reading process blocks.

Semaphores are essentially integer counters used to control access to shared resources by multiple processes. Each semaphore is associated with a location in memory whose value can be tested and set atomically by more than one process. Depending on the result of the test operation, the current process may block until the semaphore's value is changed by another process. The set operation always succeeds and adds a given value (positive or negative) to the current value of the semaphore. Semaphores can be used to implement critical regions. Binary semaphores can have only two values (0 and 1) and therefore their functionality is equivalent to a mutex. To avoid starvation, semaphore implementations use a FIFO queue for the blocked processes.

Shared memory allows processes to communicate via memory mapped within their virtual address spaces (not necessarily at the same address). This is the most efficient IPC because it avoids data copying and kernel intermediation. Before using shared memory, each process must attach to a shared memory segment using a system call. For synchronization, processes use other IPC mechanisms, e.g. semaphores, or rely on atomic operations and the memory consistency model implemented by the underlying architecture.

Most hardware platforms provide sequential consistency at the ABI level although internally often use weaker models for better scalability and performance. Sequential consistency means that there is a global order on all memory operations and it is consistent with the order specified by the program code. The memory model specifies the contract between the software and hardware in the context of three aspects: (1) ordering of loads and stores, (2) atomicity of loads and stores, and (3) store visibility (to loads). The memory model defines what can be read from memory under what conditions. To prevent unwanted reordering, processors provide special instructions: load-acquire (no memory operations can be moved before it), store-release (no memory operations can be moved after it), and full-fence (no reordering across it). IA-32 and AMD-64 implement a relatively strong memory model: all stores are store release (stores are never reordered relative to other stores), all loads are normal loads (can be reordered), and the LOCK instruction creates a full fence.

All correctly synchronized C/C++ programs are sequentially consistent. This is because in such programs all shared accessed are guarded by a mutex. Mutexes use load-acquire and store-release to prevent reordering and provide mutual exclusion for global serialization. The recent standard for C/C++ memory model [33] states that incorrectly synchronized programs are not guaranteed any well-defined semantics.

Atomic operations are an important synchronization primitive. Usually, certain basic memory access operations, such as reading an aligned byte or machine word are guaranteed to be atomic. This is the case on x86 [91] which also supports bus locking for performing such operations as read-modify-write. Most processors implement some variant of the atomic compare-and-exchange instruction. This instruction is commonly employed by *spinlocks*. Spinlocks serialize access to shared data in a non-blocking manner. To acquire a spinlock, a thread waits in a loop, repeatedly executing compare-and-exchange on a certain memory location. Since spinlocks use busy waiting, they are efficient if processes hold the lock for a short period of time and there is not too much contention.

Because of the cache coherence protocols, on multi-processor machines, an important consideration is *false sharing*. When a process periodically accesses data that will never be altered by another process, but that data shares a cache block with data that is altered, the caching protocol may force the first process to reload the whole unit even though that is not necessary. The caching system is not aware that there is no logical sharing and treats the situation as actual sharing, thus imposing high overhead on the system bus for the coherence traffic. To avoid false sharing, unrelated data structures are placed at proper distances in memory.

Other IPC mechanisms offered by state-of-the-art OSes are *network sockets* and *remote procedures*. A network socket is an endpoint of a bidirectional flow across a computer network based on the Internet Protocol (IP). Sockets enable delivery of data packets to appropriate processes using IP addresses and port numbers. Each socket is associated with a transport protocol (e.g. UDP, TCP). To establish socket communication, one process plays the role of a server that accepts incoming connections while the other process is a client.

Remote procedure call (RPC) systems, implement network/local protocols that allow a process to execute a subroutine in another address space (commonly on another host). The programmer does not explicitly implement this interaction – the code is essentially the same as for a regular procedure call. In the object-oriented context RPCs are called remote method invocations. Operating systems often optimize the protocols for the single-host case (e.g. Solaris doors [111], Sun ONC/RPC [151]). RPCs are based on a client-server model. A server registers an entry point (function address) and binds it to an external name. Function parameters are marshalled manually as system languages (C/C++) do not support reflection. Marshalling is necessary because of the possibility of architectural differences (ONC/RPC uses the eXternal Data Representation format).

2.2.2 Cross-Runtime Communication and Coordination

Managed runtimes for general-purpose languages such as Java, C#, and Python, define type-safe abstractions corresponding to some of the OS-level IPC primitives. Safe RPCs, channels, and object serialization are examples of type-safe language/library support for MRE-level, portable IPC. Their main goal is to enable exchanging objects and/or synchronization between isolated application components while maintaining the abstractions and safety guarantees of the underlying high-level programming language. Common design tradeoffs inherent to MRE-level IPC include preserving isolation, type-safety, and transparency while delivering low latency and high throughput. Some IPC protocols available in extant MREs target runtimes for a single language (e.g. Java RMI) while other ones support heterogeneous runtimes (e.g. CORBA for cross-language RPC).

There are essentially two general approaches to implementing MRE-level, type-safe IPC in state-of-the-art managed runtimes: *top-down*, where multiple applications are executed in a single process with software isolation between them, and *bottom-*

up, where multiple applications are executed in distinct OS processes. The top-down approach offers weaker (software-only) isolation guarantees, does not leverage OS-level IPC, and duplicates the OS mechanisms at the MRE level. In contrast, the bottom-up approach has stronger OS- and hardware-assisted cross-process isolation and can readily take advantage of existing standardized OS-level IPC facilities.

Extant MRE systems that support type-safe IPC and take the top-down approach fall into three categories:

- multitasking virtual machines, such as the MVM [53] and Singularity [89] that enable message passing communication via channels/links,
- shared-nothing runtimes for concurrent real-time languages, such as Erlang [8], Occam [117], and Limbo [63] that build on the algebra of communicating sequential processes [81] and support point-to-point message passing for lightweight processes, and
- multi-application runtimes/operating systems supporting shared memory, such as KaffeOS [10] and SPIN [27].

To date, the bottom-up approach has been mostly used in the context of distributed systems that can be classified into two groups:

- distributed RPCs for homogeneous language runtimes, like Java RMI, and heterogeneous runtimes, such as CORBA [50], Thrift [143], Protocol Buffers [129], and
- distributed shared memory/single system image systems, such as UPC [64], cJVM [6], MultiJav [41], and X10 [40], that support non-uniform transparent shared memory (global address space) across a cluster of computers.

Some systems providing message passing use synchronous primitives with rendezvous semantics [53] that require the sender and the receiver to use a specific channel at the same time for message delivery to take place. Other systems use asynchronous message passing which lets the sender continue execution once the message has been copied to an internal buffer (the receiver can retrieve the message at a later time).

Shared memory communication, while being more transparent and efficient than explicit message passing, necessitates defining a memory model and thread/process synchronization mechanisms. In type-safe high-level languages, the memory model tends to be more complex because safe semantics must be given even to programs that use synchronization improperly. The Java Memory Model (JMM) [75] is notorious for its complexity, which especially applies to rules for the $final$ and $volatile$ modifiers. JMM guarantees sequentially consistent semantics only to programs that are properly synchronized (i.e. those that do not contain data races). A data race occurs when multiple threads can access the same object field at the same time and at least one of them performs a write. Java provides monitor synchronization for mutual exclusion and memory access serialization (preventing reordering). Monitor entry has load-acquire semantics (downward fence) while monitor exit has store-release semantics (upward fence).

Single image systems (SSI) for Java [6, 41] extend the notion of the JMM (i.e. release consistency) into a cluster of machines. For transparent local/remote execution, threads use distributed stacks. Two approaches to code locality are: method shipping (when code is moved to a thread) and thread migration (when threads move to the code). All SSI systems use globally unique identifiers for objects in the heap.

Operating systems written in high-level languages, like SPIN [27] and Singularity [89], do not rely on protection domains for isolation but provide software-based safety based on the type-safety of the underlying language/runtime. Thus, process-kernel interaction has the cost of a method call and does not impose the overhead of switching between the kernel and the userland. In addition, no hardware support for the privileged mode and virtual memory is required. Efficient communication/event dispatch encourages the microkernel approach to the design of OS services and subsystems. The kernel has fine-grain interfaces, which promotes extensibility (monolithic kernels use coarse-grain interaction to amortize the cost of system calls). High-level languages also make automatic system verification more feasible and effective.

However, despite these software engineering benefits, in practice, OSes tend to rely on C/C++ monolithic kernels because of their better overall performance, lack of GC overhead, better predictability, and more control over the hardware, especially when writing device drivers.

Several systems mentioned in this section are worth discussing in more detail. Specifically, we overview the MVM, KaffeOS, and Singularity as representative examples of state-of-the-art IPC within a single OS process, as well as CORBA and Thrift to show the recent evolution of cross-process cross-language RPC.

The Multitasking Virtual Machine [53] extends the Java VM to run multiple Java applications within a single JVM process. The system implements lightweight isolation between the applications (tasks). Its main goal is improving JVM scalability by sharing as much runtime data/state as possible. This reduces the VM startup/initialization time (no bootstrap class loading) and memory footprint (shared data structures and class representations). To implement cross-task isolation, the MVM introduces changes to a number of runtime components, including GC, dynamic compiler, class loader, byte-code interpreter, and native code framework. Tasks cannot directly share objects. They use Java serialization and the MVM links to communicate via message passing. Links do not buffer messages – send/receive is a rendezvous point. The MVM provides the API for task creation, termination, and link setup. User-supplied native code is run in a separate OS process in order to prevent possible task interference at the C/C++ level (the JNI is implemented using OS IPC). For the Java code and core native libraries, the MVM uses software-only isolation.

Singularity [89] builds the whole system stack (the OS, managed runtime, and applications) from layers written in a high-level, verifiable language. The system has three main architectural features: software-isolated processes, contract-based channels for message passing, and manifest-based programs. Singularity focuses on dependability and correctness as a way of dealing with OS kernel vulnerabilities and faulty device drivers. Processes execute in one address space and protection is achieved by the language type-safety. No data can be shared between processes. Sending a message over a channel entails the transfer of exclusive data ownership (the compiler enforces the constraint that after sending, objects are not reused by the sender). The code is sealed, there is no dynamic class loading. Programs are defined by manifests that specify dependencies, resources, capabilities, and runtime properties for static verification. Channel contracts consist of message declarations and protocol states. Singularity uses a microkernel approach (device drivers, file systems, and OS extensions execute as processes). The exchange heap enables passing messages while avoiding copying and serialization (which applies both to channels and zero-copy I/O). However, direct object sharing is not supported. The kernel and processes use separate GCs. Channels are asynchronous with bounded FIFO queues.

KaffeOS [10] implements the abstraction of OS processes in a JVM to enable execution of multiple Java programs in a single VM process. The system defines user and kernel boundary and implements resource management with detailed accounting. KaffeOS strives to share as much runtime data and classes as possible while providing isolation for processes. Object references can cross the user/kernel boundary but no direct pointers are allowed across the heaps of different processes. To share objects, processes in KaffeOS must allocate a shared heap. The system imposes two restrictions on how shared objects may be used. First, there is no pointers from a shared heap to any private heap. Second, the size of a shared heap is frozen after its allocation and initialization. There is a dedicated class loader for each shared heap. All sharing processes are charged in full for a shared heap.

CORBA [176] is an Object Management Group standard for object-oriented communication across heterogeneous platforms in a distributed environment. CORBA specifies an architecture for location-transparent RPC for distributed objects in a language- and platform-independent way. The RPC employs the Interface Definition Language (IDL) that describes object interfaces. Messages are sent over the Object Request Broker (ORB) which is responsible for marshalling and transport. The IDL compiler creates IDL stubs (for the client) and IDL skeletons (for the server) that enable transparent static RPCs (compile-time binding). For dynamic (run-time binding) RPCs, CORBA uses the Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI), capable of service discovery at runtime. The clients use the IDL registry (interface repository) to discover the available RPC endpoints. CORBA supports synchronous, deferred synchronous, and asynchronous calls. The specification defines an implementation-neutral object model with interface-based inheritance and no polymorphism. The type system comprises basic values, object references, and complex values (structures, sequences, and unions).

Thrift [143] is a fast, scalable, lightweight RPC across languages in a distributed system. It has been developed as an alternative to CORBA to overcome some of its limitations, such as complex architecture, high overhead, and poor scalability. Thrift is an RPC library and a set of code-generation tools. The system defines a language-neutral interface specification, and generates client/server stubs/skeletons for a number of different programming languages (including Java, Python, and C++). Only static calls are supported. Thrift uses versioning which allows augmenting data types in RPC argument lists without service interruption. The type system consists of base types (primitives such as integers, booleans), structures, and containers (list, map, and set). They map to native/builtin types in each language. Thrift supports exceptions and asynchronous calls.

2.2.3 Limitations

Mainstream managed runtimes for general-purpose programming languages still considerably lag behind the OS-level IPC in the scope of supported mechanisms and/or performance. For instance, shared memory has not yet been adopted to safe languages despite being supported by OS for decades. The emergence of multi-core machines makes managed runtime co-location more and more common, thus rendering type-safe shared memory an important runtime service. In addition, extant, safe RPC mechanisms offer poor performance in the co-located case due to data structure copying/serialization, which could be avoided by using shared memory. Thus, incorporating object sharing to managed runtimes could improve both the programming model and performance.

The design of cross-runtime memory management in modern object-oriented programming languages needs to incorporate at least some of the following goals and features:

- **Cross-language communication.** Since each language has its own unique combination of performance, productivity, and library support, different software components are often implemented in distinct languages. Such components need to communicate and/or share data.

- **Cross-runtime isolation.** Preserving fault and resource isolation between components is key to the overall system robustness and its ability to stop failure propagation at the component boundaries.

- **Exploiting co-location.** Administrators increasingly co-locate multiple components on a single machine to better utilize multi-core and multi-processor shared-memory platforms via thread-level parallelism. Optimizing distributed communication protocols for the local case by taking advantage of shared memory can improve throughput and latency significantly.

- **Serialization avoidance by sharing.** The most expensive part of communication via message passing in type-safe IPC systems is data structure serialization/marshalling. This process typically involves object graph traversal, its inspection via reflection, and encoding into a byte stream. Serialization is difficult to parallelize, does not scale well, and can degrade throughput and latency by orders of magnitude. Direct object sharing avoids this overhead.

- **Type-safety and garbage collection.** Both message passing and shared memory systems must guarantee type/memory safety and provide modern garbage collection. Concurrent and on-the-fly GCs are most suitable for multi-core architectures and applications that require low latency and scalability.

- **Transparency.** For the programmer's convenience and productivity, modern cross-runtime communication schemes must be blended into the runtime type system, builtin types, and language constructs. For example, objects residing in the shared memory should behave as regular objects with respect to synchronization, method calls, field access, and other runtime services.

- **Language-neutral object model.** To communicate across languages, a channel, an RPC system, or a shared heap must define and use a language-independent object model that can be mapped to native object models in each language. This object model must strike a balance between being too narrow (easy to map but inconvenient and inefficient memory use) and too wide (may be difficult to map all types to all languages but more fine-grain control over memory use).

- **Lightweight and simple.** Complex systems are difficult to analyze and optimize. Therefore, recent RPC systems, such as Thrift, put emphasis on simplicity (the key motivation for Thrift was the heavyweight and overdesigned CORBA).

- **Easy to use by programmers.** RPC systems typically employ an interface definition language (IDL) that is compiled into stubs and skeletons. This approach necessitates keeping the IDL schema consistent with the client/server implementation in a specific programming language. In addition, programmers have to learn the IDL syntax, which tends to vary significantly between RPC systems.

Schema-less RPC/sharing (similar in spirit to recent schema-less databases) can alleviate the programming burden caused by IDLs.

- **Scalable.** More and more commodity systems have multi-core CPUs, and are equipped with large main memory. Cross-runtime communication systems must therefore scale in the number of runtimes and the shared data size.

- **Loosely-coupled architecture.** To make systems fault-tolerant and flexible, individual subsystems must avoid excessive interdependencies and centralized control. Containing failures to a single component and architecting systems to have no central point of failure are two approaches commonly used to increase system dependability.

- **Easy to evolve.** To enable fast prototyping and short software development cycles, RPCs need to permit frequent changes in the meta-data/IDL schema, preferably without stopping the deployed system. For example, versioning in Thrift significantly increases IDL elasticity.

Although many of the above-mentioned design goals have been to some extent incorporated in recent RPC systems [143, 89, 40], and some RPC technologies are mature and widely-used [176], state-of-the-art cross-runtime communication still needs to evolve to meet the requirements of modern applications and take better advantage of the underlying software/hardware infrastructure. Currently, the key limitations are:

- Operating system support for shared memory, although standardized by POSIX for decades, is not leveraged by programming languages used in production today. As a result, managed runtimes are unable to optimize communication on a local machine and always use high-overhead distributed protocols.
- Extant object sharing systems are limited to a single language and a single operating system process. We are not aware of any type-safe, managed runtime that supports cross-language shared memory where runtimes are run as separate processes. In consequence, multi-runtime multi-language systems that become more and more common can only communicate by message passing, which is suboptimal in the co-located case.
- Software-based isolation between components in state-of-the-art object sharing systems provides insufficient guarantees while complicating system design. Duplication of the resource protection and management already implemented in the operating system and hardware adds engineering effort while being less reliable and potentially incompatible with the policies implemented in the kernel.
- Cross-language RPC systems offer poor performance in the co-located case because of unnecessary serialization and copying. This is caused by the lack of support for shared memory in extant managed runtimes.

- Safe languages today offer message-passing interaction only thus limiting the programming model to channels and RPC. Object sharing systems cannot be efficiently implemented using state-of-the-art cross-runtime object-oriented communication mechanisms.

- The design of systems supporting sharing is too complex (top-down approach). RPC systems are heavyweight and use IDL schemas.

The systems described in Chapters 6 and 7 address these limitations by using OS support for shared memory to provide type-safe, transparent object sharing across homogeneous and heterogeneous runtimes co-located on a single machine.

Chapter 3

Efficient Compaction by Mapping: Improving Intra-Runtime Memory Management Performance Using Virtual Memory

In this chapter, we describe an approach to improving the performance of intra-runtime memory management by using OS support for virtual memory. Specifically, we discuss the design and implementation of a parallel and concurrent compacting collector that leverages page mapping operations. The collector exploits the observation that unreachable objects in the heap form clusters that can be effectively managed at the page granularity. Such clusters can be compacted into a new area in the virtual memory by page remapping. This avoids expensive object moving and pointer adjustment while achieving a high degree of compaction. Using page remapping allows the collector to outperform extant compactors significantly.

3.1 Introduction and Motivation

Modern systems are increasingly complex implementing multi-layered software stacks and employing more and more processing cores, in order to support a vast diversity of applications ranging from multi-media and software development to web-services and distributed gaming (among others). To extract high performance from such systems, it is vital that the layers of the software stack cooperate efficiently to make the most of the underlying hardware resources.

Two layers common to most extant systems are the operating system (OS) and the managed runtime environment (MRE) for portable, type-safe applications (e.g. those written in Java or the .Net languages). A subsystem that can significantly impact performance and that has the potential for better OS-MRE interaction is memory management.

MREs typically implement garbage collection (GC) to simplify the programming model for developers. Modern managed runtimes increasingly employ parallel and concurrent collectors [25, 1, 67, 46, 101] to maintain scalability, as multi-core architectures and multi-threaded applications become more and more commonplace. Moreover, state-of-the-art MREs often use compaction to eliminate heap fragmentation and enable fast linear object allocation [99].

Extant GCs achieve compaction by moving live (reachable) objects. This involves copying and pointer adjustment, both of which are increasingly expensive because of the growing processor-memory performance gap, and can adversely impact application performance [32]. To address this limitation, we investigate a new approach to parallel/concurrent, compacting GC in which the MRE uses standard, portable, unprivileged virtual memory operations, supported by the OS interface, to eliminate object moving. We design and implement the Mapping Collector (MC), which leverages page mapping to compact free space instead of compacting live space.

MC exploits the widely-known phenomenon that objects with similar lifetimes tend to exhibit spatial locality in the heap [167]. In particular, we find that dead objects often occur in large clusters. MC exploits this behavior to reclaim heap space at the granularity of virtual pages. The collector trades off a small heap space overhead for fast, inexpensive compaction. In practice, this space overhead is below 6% on average and MC can additionally bound it by an infrequent fall-back to state-of-the-art moving compaction. MC maintains the simplicity and low cost of a non-moving collector while providing effective compaction in the common case.

We implement both stop-the-world and concurrent MC in a generational garbage collection framework within the open-source HotSpot Java Virtual Machine. MC is applicable to both server systems (which typically employ concurrent GC to reduce pause times at the cost of resource over-provisioning [85]) and deskside systems (which tend to use stop-the-world (STW) GC because of its simplicity, higher throughput, and more efficient use of the underlying resources [85]). Our experimental evaluation using a multiprocessor indicates that MC significantly increases throughput and scalability as well as reduces pause times, relative to state-of-the-art, parallel and concurrent compactors.

Prior work on compaction has focused on both partial elimination of object moving [57, 85] and reducing the number of GC phases [101, 1, 67, 137]. MC leverages MRE-OS interaction to improve over these approaches by eliminating copying altogether. Virtual memory support for GC has been shown to be effective in other contexts including preventing collector-induced paging [77, 175, 179, 80, 174] and reducing the space overhead of copying collection via page unmapping [101, 136]. Unlike previously reported systems, MC employs virtual memory unmapping as a primary and sole technique to implement STW/concurrent compaction in a modern MRE. MC achieves almost the same effect as object moving but avoids object copying and thus improves GC performance while imposing a small space overhead. MC is a nearly-single-phase compactor while extant compacting GCs require at least two phases.

In the next sections, we overview the design and implementation of MC (Section 3.2), present the results of our empirical evaluation (Section 3.3), discuss related work (Section 3.4), and conclude (Section 3.5).

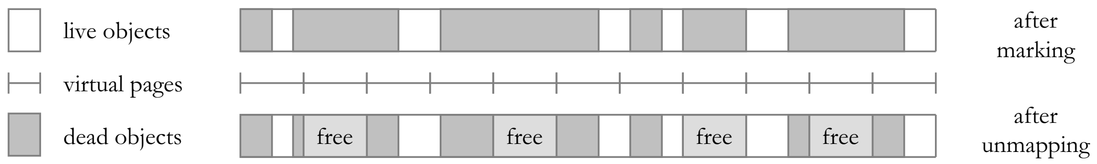


Figure 3.1: Page-based free space reclamation in MC. Virtual pages fully contained in dead clusters are returned to the OS.

3.2 Design and Implementation

MC exploits the widely-observed statistical property that unreachable objects tend to cluster together [167] and form contiguous dead regions in the heap. Our experimental analysis of modern Java programs (which we present in Section 3.3) confirms this property and reveals that clusters of dead objects are often sufficiently large to make their reclamation via virtual page unmapping practical.

Extant garbage collectors do not take advantage of the level of indirection offered by virtual memory and compact the heap by moving objects and updating pointers. MC remaps the free space into a contiguous region in a newly allocated area in virtual memory. This approach is simpler and more efficient than object copying and pointer adjustment. It enables nearly-single-phase compaction, while state-of-the-art compactors comprise at least two phases. In addition to marking, MC requires only a single traversal over the liveness bitmap (whose size is 3% of the heap).

To achieve portability, MC relies only on standard virtual memory operations [130], such as page mapping and unmapping, that are available for (unprivileged) processes as part of an operating system interface (system calls) on most modern platforms. We note that it is not sufficient to rely on the OS paging mechanism to swap out unreachable, never-accessed pages, and completely avoid garbage collection. Periodic page unmapping is necessary to free the associated OS resources (e.g. the swap space) – otherwise they are not freed until program termination.

Since virtual page granularity is larger than the unit of allocation (most objects are small) and because of the page alignment requirements of modern systems (e.g. 4KB in Linux), MC incurs a certain heap space overhead, which we evaluate in detail herein. We find that the size of the uncollected free space is modest in most cases and can be bounded via an infrequent fall-back to perfect compaction (Section 3.2.4).

By remapping free space into a new area in virtual memory, MC consumes increasingly more address space as subsequent compactions occur. This phenomenon, however, is not a problem on modern 64-bit architectures that have practically inexhaustible virtual address space at their disposal.

Like most state-of-the-art compactors, MC is designed for a tenured generation in a generational [155, 99] garbage collection system. In the young generation, normally a copying collection is used as it is more efficient than compaction if the expected percentage of live objects is low. The cost of collecting the tenured generation typically dominates GC performance.

The tenured generation contains objects with relatively long lifetimes and its allocation rate is relatively low (compared to the young generation). Thus, the expected rate at which new dead clusters appear is low and address space usage remains tolerable even on 32-bit architectures (which we have verified experimentally).

MC consists of a single parallel marking phase (which imposes the dominant cost of the collector) and a series of operations for unmapping and updating auxiliary data structures. Unmapping occurs immediately following marking and has a cost proportional to the size of the liveness bitmap (which is approximately 3% of the mapped heap size). Thus, MC is a nearly-single-phase compactor.

MC can be implemented as both STW and concurrent compactor. During unmapping, MC does not access live objects at all, and therefore can execute concurrently with the application without the need for any synchronization. This significantly simplifies the design – note that moving compactors require OS support to handle concurrent mutations to the moved objects.

While STW compaction is triggered only upon heap space exhaustion, concurrent compaction is initiated early, when a certain heap occupancy is reached (typically around 70%). This is necessary to guarantee space for allocation while the compaction progresses in the background.

3.2.1 Stop-the-World/Concurrent Marking

The marking phase identifies all reachable objects in the heap and records the starting and ending words for each live object in the liveness bitmap. Both STW and concurrent marking can be used with MC.

State-of-the-art STW parallel marking [67, 85] uses work stealing for dynamic load balancing. The root set is assigned to the marking GC threads in a round-robin fashion. Whenever a thread becomes idle, it steals a group of references from another (randomly-selected) thread. Each thread maintains a local marking stack (for depth-first search). To ensure that each live object is processed exactly once, marking GC threads claim objects atomically. GC threads coordinate marking termination via barrier synchronization.

State-of-the-art concurrent parallel marking [127, 85] consists of three sub-phases: STW initial marking, concurrent marking, and STW final marking. Initial marking suspends mutators to record all objects directly reachable from the roots. Concurrent marking resumes mutators and marks a transitive closure of reachable objects. Due to concurrent pointer updates some live objects might be left unmarked. Therefore, the algorithm keeps track of all pointer updates by leveraging a card table mechanism of a generational GC system. Final marking suspends the mutators and repeats marking from the roots treating modified pointers as additional roots. Final marking is typically short as it skips the already-marked objects. Each sub-phase can be executed by multiple parallel GC threads.

3.2.2 Stop-the-World Unmapping

STW MC performs unmapping when the mutators are suspended. The goal of the unmapping scan (which amounts to a traversal over the liveness bitmap) is to return reclaimable pages to the OS and to compute the total size of free space available in dead clusters.

MC performs the unmapping scan in parallel. Since the size of the liveness bitmap is relatively small, we do not employ dynamic load balancing. MC statically partitions the bitmap into nearly-equal-sized chunks (as many as the number of GC threads). A boundary between two adjacent chunks is the first word of a live object. Thus, the subdivision does not hinder our ability to detect regions suitable for unmapping. No synchronization is necessary between the parallel threads since we divide the marking bitmap between threads at live object boundaries and, as a result, no conflicts can occur.

MC invokes the unmapping system calls in parallel which is more scalable than serialized unmapping, especially given that pages returned to the OS by different GC threads belong to disjoint virtual memory areas. OS kernels that support fine-grain locking in the memory management subsystem can likely handle such concurrency with little contention.

Figure 3.1 illustrates how MC reclaims free space on a virtual page basis. The unmapping scan identifies unreachable regions and unmaps their fragments that fully cover the underlying virtual pages. Since MC does not move objects, the freed areas never contract, and unmapped pages remain unused. The space overhead tends to improve over time as small dead fragments scattered across the heap assemble into larger clusters that MC can later unmap.

MC maintains a page bitmap to track heap pages that are currently unmapped. Its size is approximately 0.003% of the used address space (1 bit per 4KB). Without this additional data structure, the performance of long-running applications that exhibit high object turnover in the tenured generation may degrade. The unmapping scan traverses over the liveness bitmap which has a size of approximately 3% of the address space currently used by the heap. This includes the unmapped areas. Therefore, to keep the cost of the unmapping scan proportional to 3% of the heap size (not the used address space), MC must distinguish between mapped and unmapped regions. With this enhancement, MC can traverse (and clear) the liveness bitmap only partially (skipping the unmapped regions). In addition, this reduces the number of unmapping system calls (as we do not unmap the same clusters multiple times).

Once the unmapping scan is complete, MC expands the heap by the total size of the newly-discovered free space (not the total size of the newly-unmapped pages) in the heap (to enable identical behavior as and a fair comparison to perfect compacting collectors). The space overhead of MC then, is the size of this expansion minus the total size of the pages that MC has unmapped in the current collection cycle.

3.2.3 Concurrent Unmapping

In concurrent MC, unmapping takes places after resuming the mutator threads. MC first traverses over the liveness bitmap, finds dead clusters (their addresses and sizes are stored in the cluster array), and clears the bitmap. During the bitmap traversal, MC also computes a new object-start array, necessary in a generational GC system to locate the first object on any 512-byte card during the young generation collection [144]. Since these activities are performed concurrently to mutators, a young-generation GC might take place in the background (two collectors may execute at the same time). Therefore, MC must compute the object-start array using a separate (shadow) array. This translates to 0.2% space overhead (1 byte per 512 bytes). Next, MC suspends the mutators, and finishes the computation of the shadow array. Note that during the concurrent pass over the bitmap, new allocations might have taken place in the old generation. These new objects need to be taken into account when generating the shadow array. While the mutators are stopped, MC switches to the new shadow array and inserts filler objects into dead clusters. Card table entries (dirty/clean cards) are left intact (as no object moves). In addition, MC computes the new size of free space and resizes the heap accordingly (by the total size of the newly-discovered free space). Finally, the mutators are resumed, and free clusters are unmapped concurrently. Thus, there is one STW sub-phase and two concurrent sub-phases. Auxiliary data structures used by concurrent MC (the cluster array and the shadow array) impose additional space overhead. However, this overhead is small in practice, and, as we discuss later, is not an issue given that concurrent GC needs significantly over-provisioned heaps.

3.2.4 Bounding Space Overhead

STW MC supports space-bounded collection by falling back to perfect compaction in cases when unmapping fails to reclaim a sufficient amount of free space. In case of concurrent MC, there is no need for bounding the space overhead as concurrent MC requires significantly more heap space than STW MC (much more than the imposed space overhead). This is because concurrent GC trades pause times for space and throughput (Section 3.3.6).

STW MC evaluates whether to perform a fall-back after STW parallel unmapping. In most state-of-the-art parallel compactors, (including MC, HS, and CP), a liveness bitmap is the interface that bridges marking and the subsequent phases. Therefore, MC can directly proceed to the second phase of a conventional moving compactor without any additional processing, once it determines that a fall-back is needed.

Our current MC fall-back is the STW Compressor. The compaction phase of the Compressor is described in Section 3.4.1. An alternative solution is a fall-back to the

HotSpot compactor, but STW CP imposes a smaller space overhead and is simpler. The space-bounded MC uses two mutually-distant areas in the address space, one of which is active (and mapped) at any given point in time. The non-moving unmapping-based compaction always takes place in the currently active space. If a fall-back is needed, then all objects from the active space are moved to the other space and the roles of the two spaces are flipped (as in the Compressor). The time overhead imposed by a fall-back is the unmapping scan (the moving compaction does not benefit from this scan) and includes bitmap traversal, unmapping, filler object insertion, and object-start array computation.

3.2.5 Implementation Details

We have implemented STW MC (the unbounded and the space-bounded variant), concurrent MC, and the STW/concurrent Compressor in HotSpot [118], an open-source (GPL) high-performance Java Virtual Machine available from Sun Microsystems and written in C/C++ (source code released on 3/21/2007). The HotSpot JVM uses a generational [155] heap layout that comprises the permanent, tenured (old), and young generation. The young generation is further subdivided into eden and two equal-sized survivor spaces (called from-space and to-space). The permanent generation contains run-time meta-data for the loaded classes. The system allocates objects initially in the eden (if their size precludes eden allocation, it allocates them directly in the tenured generation). Upon space exhaustion in the eden, a copying collector [42, 67] (called the scavenger) performs a minor collection. The scavenger evacuates live objects from the eden-space and from-space to the to-space, and promotes objects that survive several minor collections (or those that do not fit into the to-space) to the tenured generation. The roles of the survivor spaces exchange after each minor collection. When space in the tenured generation is exhausted, a major collection (compaction) takes place. The parallel STW compactor currently available in HotSpot is described in Section 3.4.2. GC threads in HS are schedulable kernel threads. HotSpot assigns each generation a contiguous region in the virtual address space and maps only the currently used portion.

We implement STW/concurrent MC as a parallel compactor in the tenured generation. Both STW and concurrent MC use STW parallel marking. We reuse and simplify the marking phase of the STW parallel HotSpot compactor (MC does not require per-chunk summary data). We increase the distance between generations in virtual memory to reserve address space for page remapping.

MC compacts the young generation (which is much smaller than the tenured generation) by object moving and pointer adjustment. This compaction, however, is not part of the major collection. It takes place as an epilogue of a failed minor collection. Consequently, MC does not need to update any pointers during major collections (unlike HS and CP).

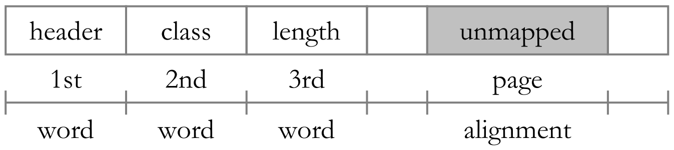


Figure 3.2: The format of a filler object. First three words form the header of an array object. The page-aligned part of the rest of the cluster is subject to unmapping.

Since the scavenger uses a card table to find roots during minor collections, the unmapping scan in MC must compute an offset of the first live object for each 512-byte card (the object-start array). This additional processing is concomitant to the dead-cluster unmapping and does not require a separate pass.

Free regions cannot be entirely unmapped as the scavenger must be able to traverse (object by object) an arbitrary subspace of the tenured generation (in search for roots) during minor collections. Therefore, we insert a filler object into every free area during each unmapping scan. Figure 3.2 depicts the format of the filler object. The type of a filler object is an integer array (`int[]`), to ensure that there are no interior reference fields for the scavenger to follow. Thus, each free region is reclaimable except for three words that are necessary for the header of a filler object. The minor GC treats filler objects as if there are live, however, since they are unreachable, the next major collection considers them to be garbage. Following the HotSpot convention, we use a single system call (`mmap`) to perform both mapping and unmapping (for the latter we employ the `MAP_NORESERVE` flag).

Concurrent MC requires a STW phase in order to atomically update the object-start array, insert filler objects, and resize the heap. We piggyback on the STW young generation collection to avoid introducing additional expensive safepoints [85]. Young generation GC is relatively frequent and a slightly-delayed STW phase is not a problem in practice.

Generational Compressor We extend the Compressor to support generational compaction, and implement it in the tenured generation. The Compressor moves objects, therefore it needs to update the pointers in the young and permanent generations upon each compaction. We use 256-byte blocks, as we have found them to be the best trade-off between space overhead and performance. The concurrent Compressor has two concurrent sub-phases, separated by a single STW sub-phase. In the first sub-phase, the Compressor computes the block-offset array (used for pointer forwarding) and the shadow object-start array. In the STW sub-phase, the system updates the shadow object-start array (to include new allocations) and sets it as the current object-start array, invalidates card tables (because objects are moved), forwards pointers in the young generation and permanent generation, protects heap pages and switches to the other semi-space. In the third sub-phase, a concurrent thread reads subsequent pages (one word per page to generate SEGV traps) to ensure that all the pages are eventually moved, and clears the liveness bitmap.

3.3 Experimental Evaluation

We empirically evaluate 6 compactors: STW HotSpot, STW unbounded MC, STW space-bounded MC, concurrent unbounded MC, STW Compressor, and concurrent Compressor. We compare these GCs in two groups, one comprising 4 STW compactors and the other comprising 2 concurrent compactors. In addition, we compare STW MC with concurrent MC to investigate the STW/concurrent tradeoffs.

Our experimental platform is an SMP with 4 processors each of which is a 2-way SMT (the machine has 8 logical CPUs). Each physical processor is a 32-bit Intel Xeon with 1MB of cache, clocked at 1.6GHz. The machine is equipped with 7GB of main memory and is running Linux Red Hat 3.4.6 with the 2.6.9 kernel. The virtual page size is 4KB. We run HotSpot 7-ea-b10 compiled with GCC 3.2.3 in the optimized client-compiler (C1) mode.

3.3.1 Benchmarks

Our benchmarks include three multi-threaded server benchmarks: VolanoMark 2.5 [156], PseudoSPECjbb 2000 [145], and Hsqldb from the DaCapo 2006 suite [54], as well as three deskside utilities (from DaCapo 2006): Xalan, Chart, and Pmd. We list the basic statistics for these benchmarks (i.e. the minimum heap size, total execution time,

Benchmark	Heap[MB]	Time[s]	GC[%]	#GCs
Chart	27	25.79	13.21	16
Xalan	31	20.39	43.48	68
Pmd	31	29.54	28.16	26
Hsqldb	100	18.62	36.48	4
Volano	33	80.25	24.41	112
JBB	174	95.62	43.02	84

Table 3.1: GC statistics for the HotSpot compactor: the minimum heap size, execution time, percentage of GC time relative to execution time, and the number of GCs. The measurements have been obtained for the minimum heap size for each benchmark.

total GC time, and the number of GCs), that we obtain using the HotSpot compactor, in Table 3.1.

VolanoMark is a standard server benchmark derived from a commercial chat server (VolanoChat), which simulates a multi-user environment with multiple chat rooms. The benchmark exchanges a given number of messages and reports execution time and communication throughput. PseudoSPECjbb is a variant of SPECjbb that executes a given number of transactions and reports execution time. The benchmark emulates a three-tier client-server system (with emphasis on the middle tier) where clients are replaced by driver threads and database storage by binary trees of objects. Hsqldb is a relational SQL database management system that supports in-memory and disk-based data storage. DaCapo employs Hsqldb to execute an in-memory benchmark that comprises a number of transactions against a model of a banking application. Xalan transforms XML documents into HTML. Pmd analyzes a set of Java classes for a range of source code problems. Chart plots a number of complex line graphs and renders them into a PDF file.

3.3.2 Methodology

Each of our experiments uses a fixed-size heap. We report total heap size, which includes the young, old, and permanent generation. Total heap size does not include auxiliary data structures as they are located outside of the heap. The young generation size is 25% of the old generation. The permanent generation is 12MB (HotSpot default). Explicit GC invocation and adaptive generation resizing are disabled. We employ 4 parallel GC threads (except for the scalability experiments where we use 1–8 threads). Survivor spaces (from-space and to-space) occupy 33% of the young generation (the remaining space is used by the eden). For concurrent MC/Compressor we start compaction when 65% of the old generation is used. Concurrent compaction uses a single concurrent GC thread.

We repeat each measurement three times and report the average result along with the standard deviation (error bars in the plots), wherever appropriate. We employ the default input size for all DaCapo benchmarks. VolanoMark is run with 44 chat rooms and performs 100 iterations in the networked mode. The server and the client are on the same machine. PseudoJBB is configured to execute 10^5 iterations against 8 (for STW GC) and 4 (for concurrent GC) warehouses.

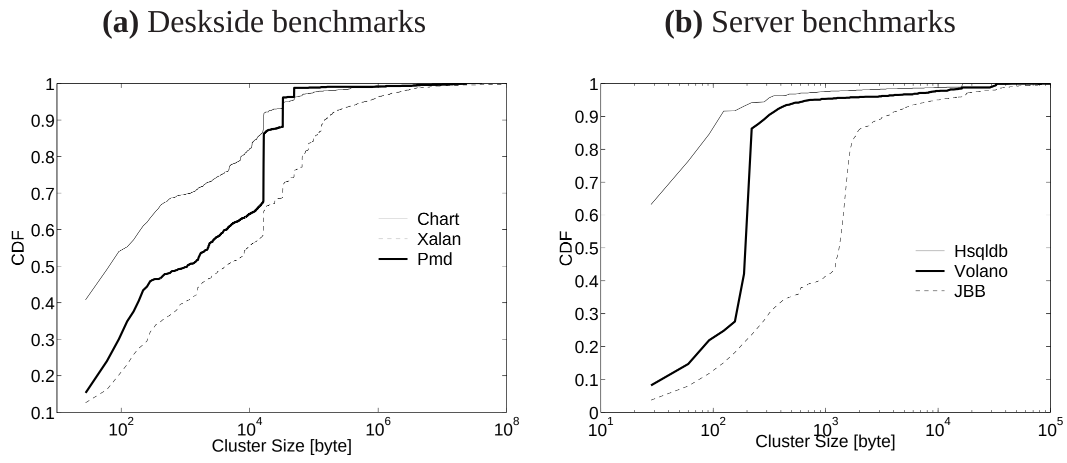


Figure 3.3: Distribution of cluster sizes for the deskside benchmarks (a) and server benchmarks (b). We report CDFs for individual benchmarks.

3.3.3 Clustering

Figure 3.3 shows CDFs for the sizes of clusters of dead objects for the deskside benchmarks (a) and server benchmarks (b), while Figure 3.4 presents summary CDFs across the benchmarks. We report data obtained for the minimum heap sizes using STW unbounded MC. Percentage of clusters greater than 4KB (virtual page size) is 24% for Chart, 52% for Xalan, 38% for Pmd, 1% for Hsqldb, 5% for Volano, and 9% for JBB. Fragmentation is higher in server benchmarks. MC achieves low space overhead for these benchmarks by reclaiming relatively few big clusters rather than many smaller ones. Average cluster size is 26KB, minimum cluster size is 28B, and maximum cluster size is 184MB.

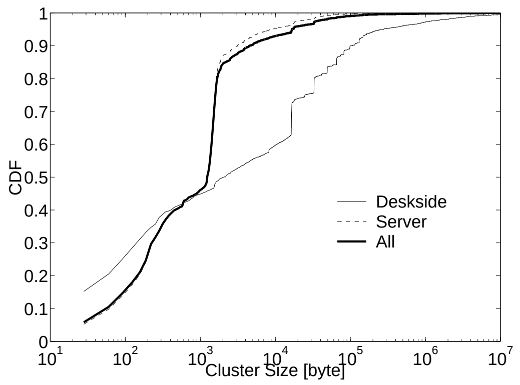


Figure 3.4: Distribution of cluster sizes across the benchmarks. We report CDFs for deskside, server, and all benchmarks.

3.3.4 Stop-the-World Compactors

We compare STW unbounded MC (UN) and STW space-bounded MC (SP) with STW Compressor (CP) and STW HotSpot (HS) in terms of memory footprint, throughput, pause times, and scalability. For SP, we employ the 10% space overhead bound in all experiments. We also investigate the impact of other bounds on the fall-back frequency and average pause times.

Space Overhead

HS and CP impose a constant space overhead of 3% (for 2KB chunks) and 1.5% (for 256B blocks), respectively. In MC, the space overhead is variable and application-specific (but can be bounded) and depends on the degree of dead-object clustering in the heap.

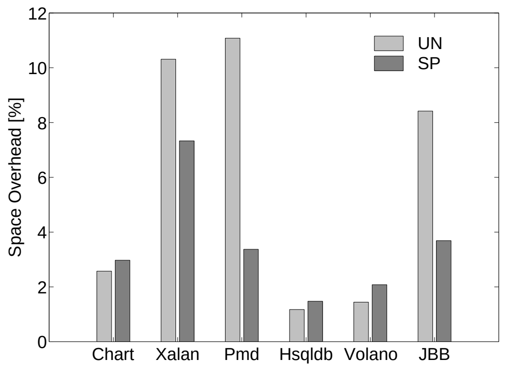


Figure 3.5: Space overhead across the heap sizes for STW unbounded MC (UN), and STW space-bounded MC (SP) with the 10% bound.

The bar graph in Figure 3.5 shows space overhead imposed by STW unbounded MC and STW space-bounded MC. For each benchmark, we report the average value across the heap sizes. The overhead is shown as a percentage of the heap size. On average, the unbounded MC imposes 5.8% overhead while the space-bounded MC (with the 10% bound) imposes 3.5% overhead.

Throughput

In Figure 3.6, we present per-benchmark graphs, each with four performance curves for a range of heap sizes. Each graph shows execution time as a function of heap size (starting from the minimum heap size).

For the minimum heap sizes and relatively to HS, UN improves throughput by up to 23.5% (Hsqldb) and by 13.3% on average. For the minimum heap sizes and relatively to CP, UN improves throughput by up to 42.1% (PseudoJBB) and by 23.3% on average.

For the minimum heap sizes and relatively to HS, SP improves throughput by up to 22.7% (Hsqldb) and by 10.9% on average. For the minimum heap sizes and relatively to CP, SP improves throughput by up to 40.1% (PseudoJBB) and by 21.1% on average.

Pause Times

Figures 3.7(a) and 3.7(b) present average and maximum pause times for UN, SP, HS, and CP. For each benchmark, we report the average value across the heap sizes.

Compared to HS, UN reduces average (maximum) pause times by up to 69.7% (78.7%) and on average by 63.4% (68.4%). Compared to CP, UN reduces average (maximum) pause times by up to 73.8% (74.4%) and on average by 66.8% (67.5%). Compared to HS, SP reduces average (maximum) pause times by up to 67.8% (76.4%) and on average by 49.3% (31.4%). Compared to CP, SP reduces average (maximum) pause times by up to 72.2% (71.7%) and on average by 53.9% (31.5%).

A commonly-employed GC metric for the evaluation of collector-imposed pauses are minimal mutator utilization (MMU) curves [43] (we discuss MMU in more detail in Section 2.1). As shown in Figure 3.8, UN achieves the highest MMU for all window sizes across all benchmarks and attains non-zero utilization for windows shorter than SP, HS, and CP. SP achieves better or the same utilization as HS and CP for all benchmarks. Since SP falls back to CP, its maximum pause time is often similar to CP. HS achieves better or comparable utilization as CP.

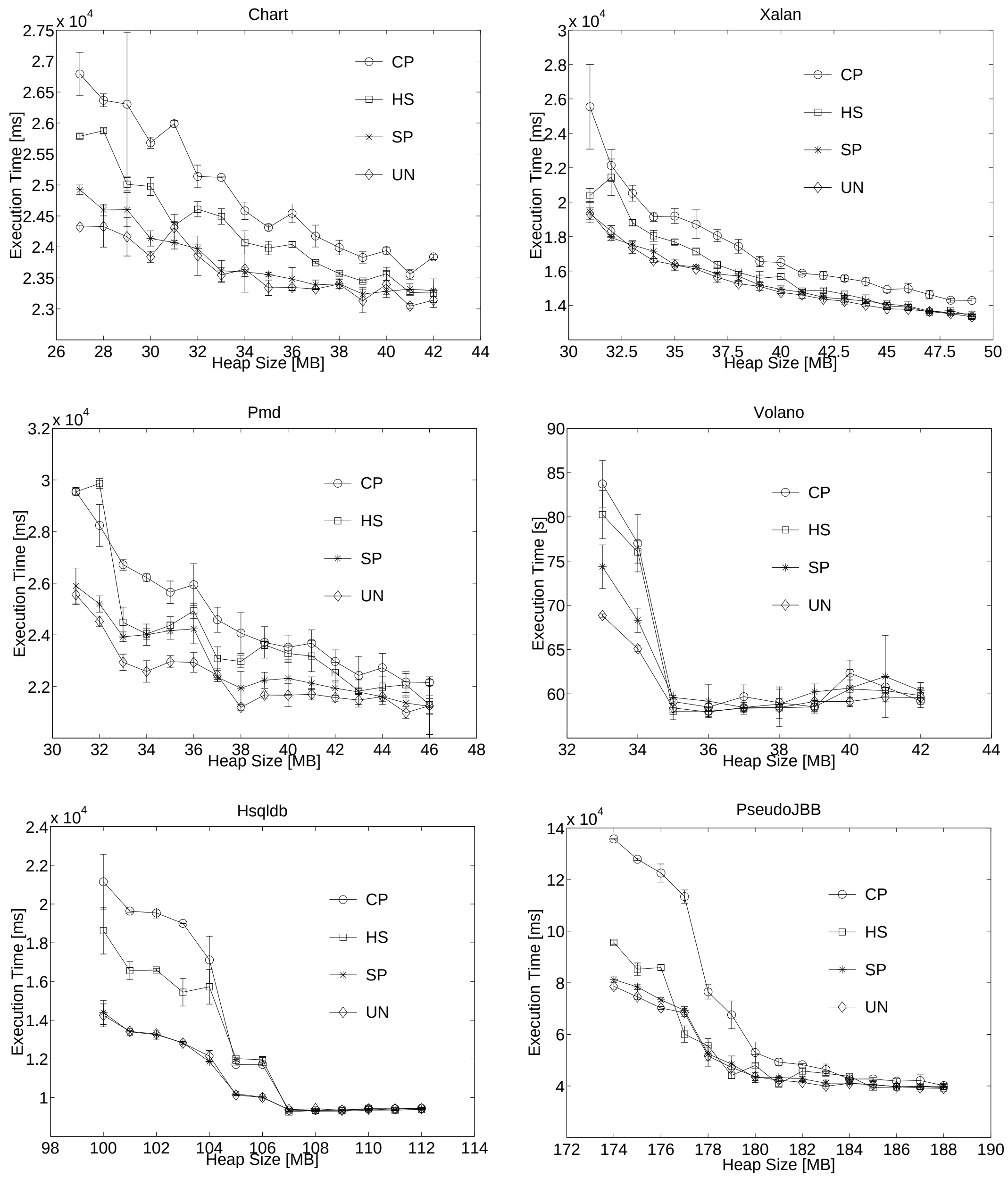


Figure 3.6: Benchmark performance (execution time) across the heap sizes for STW unbounded MC (UN), STW space-bounded MC (SP) with the 10% bound, STW HotSpot compactor (HS), and STW Compressor (CP). Error bars indicate the standard deviation across 3 runs.

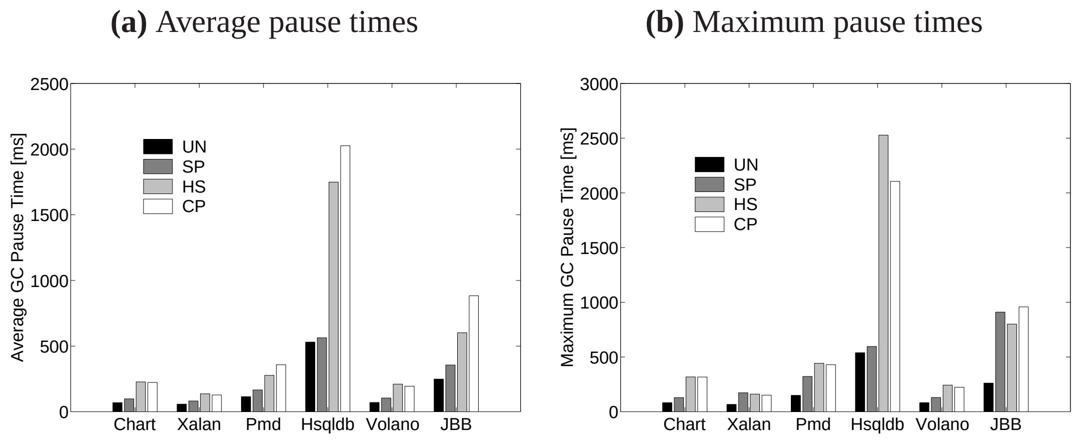


Figure 3.7: GC pause time statistics across the heap sizes for STW unbounded MC (UN), STW space-bounded MC (SP) with the 10% bound, STW HotSpot compactor (HS), and STW Compressor (CP): average pause times (a) and maximum pause times (b).

Figure 3.9 compares average (data points) and maximum (error bars) pause times for UN, SP, HS, and CP. For these experiments, we vary the number of parallel GC threads for a fixed heap size (we use the minimum heap sizes). Both UN and SP consistently decrease pause times relative to HS and CP, independent of the number of parallel GC threads. For 1 GC thread, UN reduces pauses on average by 49% relative to HS and by 61% relative to CP, while SP reduces pauses on average by 44% relative to HS and by 56% relative to CP. For 4 GC threads, UN reduces pauses on average by 61% relative to HS and by 66% relative to CP, while SP reduces pauses on average by 51% relative to HS and by 57% relative to CP. Finally, for 8 GC threads, UN reduces pauses on average by 65% relative to HS and by 66% relative to CP, while SP reduces pauses on average by 54% relative to HS and by 54% relative to CP.

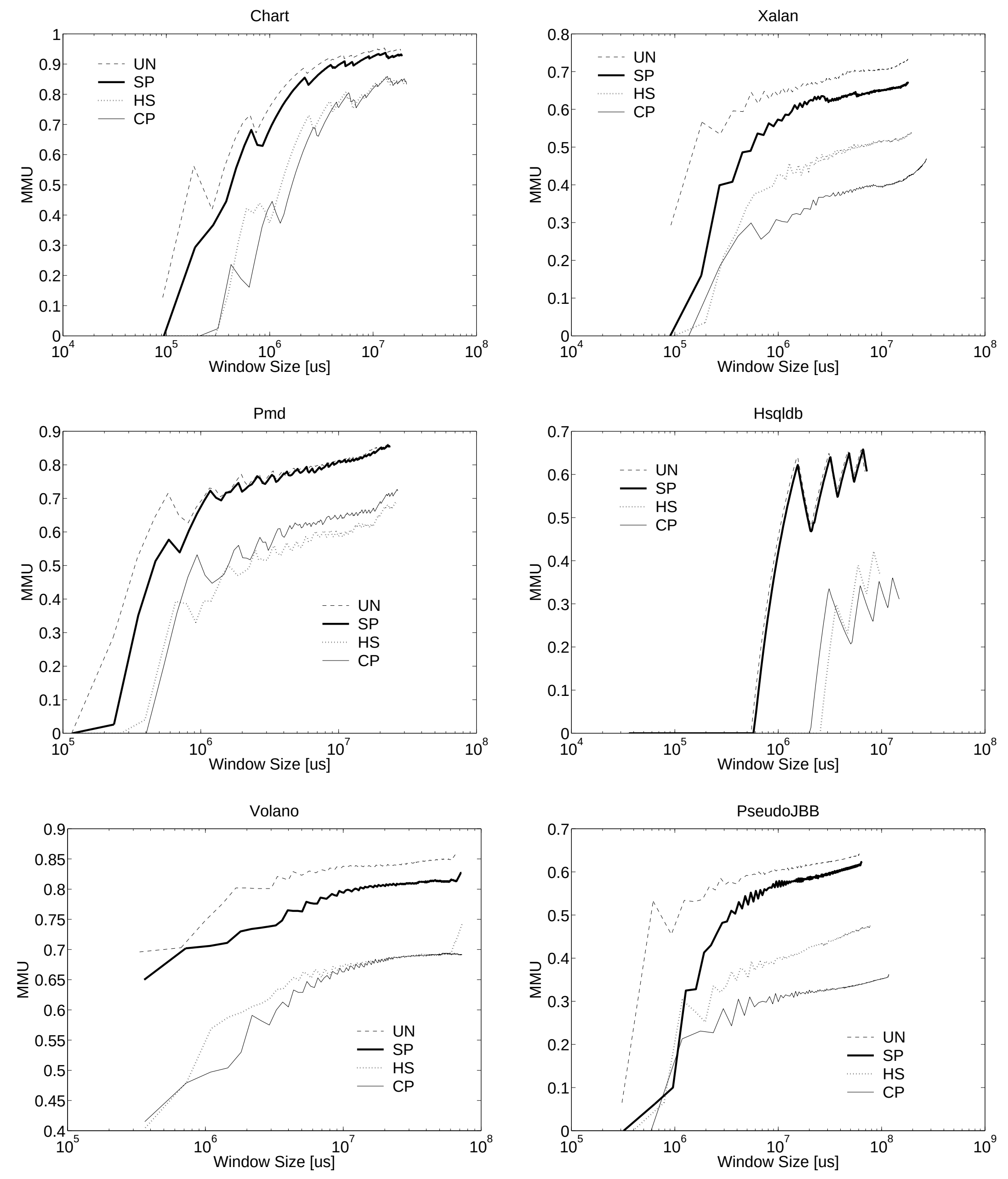


Figure 3.8: Minimum mutator utilization (MMU) curves for the minimum heap sizes for STW unbounded MC (UN), STW space-bounded MC (SP) with the 10% bound, STW HotSpot compactor (HS), and STW Compressor (CP). Window size is in microseconds.

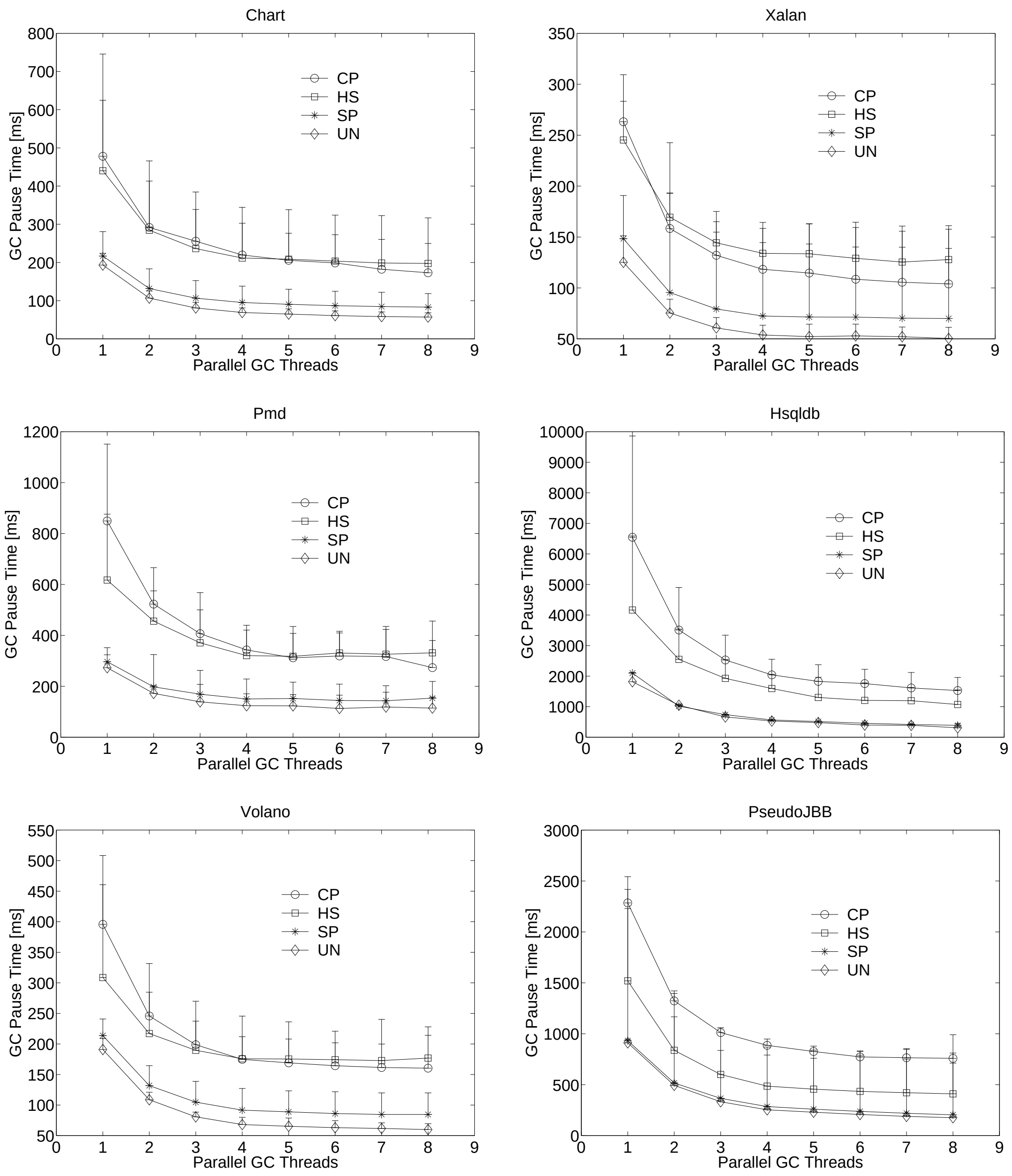


Figure 3.9: Average (data points) and maximum (error bars) GC pause times for 1–8 parallel GC threads and the minimum heap sizes for STW unbounded MC (UN), STW space-bounded MC (SP) with the 10% bound, STW HotSpot compactor (HS), and STW Compressor (CP). We report average values across 3 runs.

Scalability

Our experimental platform has four 2-way SMT processors virtualized by the operating system as 8 logical CPUs. We investigate the scalability (speedup) of UN, SP, HS, and CP in the context of both multi-processing and multi-threading parallelism. We measure the unscaled speedup – we apply an increasing number of GC threads (from 1 through 8) to a fixed-size workload and the minimum heap for each benchmark. We compute the speedup for p threads as a ratio of the average GC pause time for 1 thread and for p threads.

As shown in Figure 3.10, server benchmarks scale better (e.g. Hsqldb/UN achieves 5.9 speedup while the maximum for deskside benchmarks is 3.4 for Chart/UN). HS has the worst scalability because it computes per-chunk statistics during marking, which entails more synchronization. The plots in Figure 3.9 provide absolute average GC pause times from which the speedup graphs have been derived.

When considering only multi-processing parallelism (4 GC threads), the speedup averages at 2.86 for UN, 2.6 for SP, 2.22 for HS, and 2.49 for CP. Thus, UN improves speedup by 30% relative to HS and by 15% relative to CP, while SP improves speedup by 17% relative to HS and by 4% relative to CP.

When multi-threading is taken into account (8 GC threads), the speedup averages at 3.75 for UN, 3.19 for SP, 2.56 for HS, and 3.03 for CP. Thus, UN improves speedup

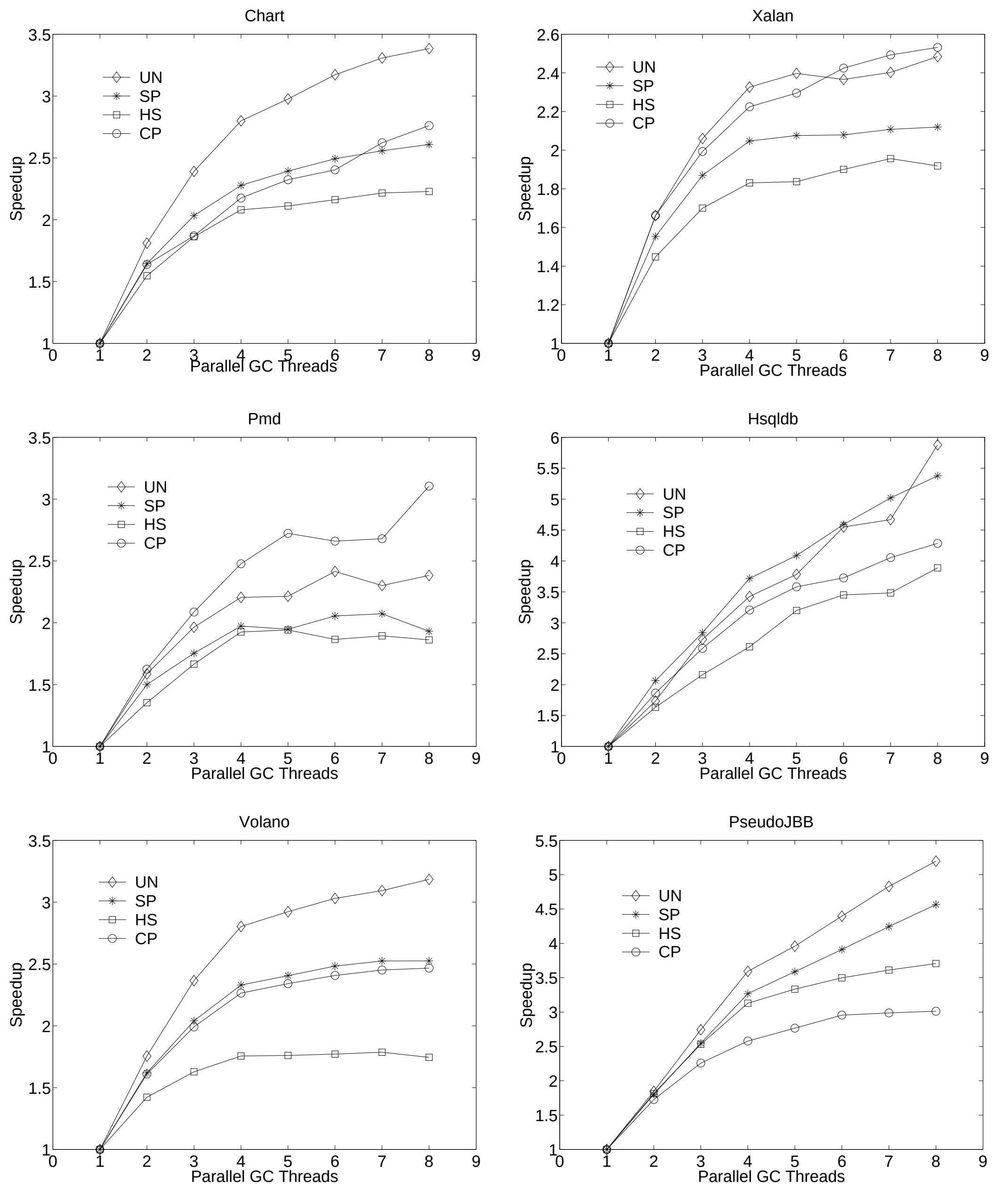


Figure 3.10: Scalability (unscaled speedup) for 1–8 parallel GC threads, fixed workload, and the minimum heap sizes for STW unbounded MC (UN), STW space-bounded MC (SP) with the 10% bound, STW HotSpot compactor (HS), and STW Compressor (CP). Speedup is computed for average GC pause times. Absolute average GC pause times are reported in Figure 3.9.

by 47% relative to HS and by 23% relative to CP, while SP improves speedup by 23% relative to HS and by 3% relative to CP.

Fall-Back Rate

SP falls back to CP if excessive fragmentation in the heap makes it impossible to reclaim a significant fraction of free space. Table 3.2 shows the rate of fall-back to perfect compaction that is necessary to guarantee a specific space overhead bound (2% to 20%). We express this rate as the percentage of GCs that need to fall back to conventional moving compaction to keep the space overhead below a given threshold.

The fall-back statistics for the minimum heap sizes indicate that even for tight bounds, relatively infrequent fall-back is necessary. For instance, in order to achieve 5% bound, on average, 6.5% collections need to trigger a fall-back (for 7% bound it is 3.7% and for 10% bound it is 1.8%). In addition, we have measured average GC pause times for different space bounds. The results for UN and SP, reported in Table 3.2, indicate that the space-bounded MC reduces pauses significantly relative to HS and CP (for all bounds that we investigate), and increases average pause times by around 20% compared to the unbounded MC.

Bound [%]	2	5	7	10	15	20
Benchmark	Fall-back frequency [%]					
Chart	12.8	0.0	0.0	0.0	0.0	0.0
Xalan	99.6	32.6	16.4	5.7	1.5	0.4
Pmd	6.6	3.9	4.2	4.1	4.1	0.0
Hsqldb	0.0	0.0	0.0	0.0	0.0	0.0
Volano	1.0	0.0	0.0	0.0	0.0	0.0
JBB	4.3	2.3	1.6	1.2	0.0	0.0
Compactor	Avg. Pause Decrease [%]					
STW CP	62.6	64.7	65.2	65.1	65.2	66.1
STW HS	53.2	55.8	56.4	56.3	56.4	57.5
Compactor	Avg. Pause Increase [%]					
STW UN	26.8	22.5	21.4	21.5	21.4	19.4

Table 3.2: GC statistics for STW space-bounded MC for different space bounds obtained using the minimum heap sizes. The first part shows fall-back frequency (GC percentage). The second part shows percentage decrease in average GC pause times relative to STW Compressor (CP) and STW HotSpot (HS). The third part shows percentage increase in average GC pause times relative to STW unbounded MC (UN).

3.3.5 Concurrent Compactors

Next, we compare concurrent unbounded MC (UN) and concurrent Compressor (CP) in terms of memory footprint, throughput, and pause times.

Space Overhead

Concurrent collection requires heap space over-provisioning to avoid the situation when allocators exhaust the heap before the ongoing background collection is complete. Therefore, space overhead is less of a problem in concurrent MC than in STW MC. Figure 3.11 shows space overhead (as a heap percentage) averaged across the heap sizes. As explained earlier, CP has a constant space overhead of 1.5%. Across the

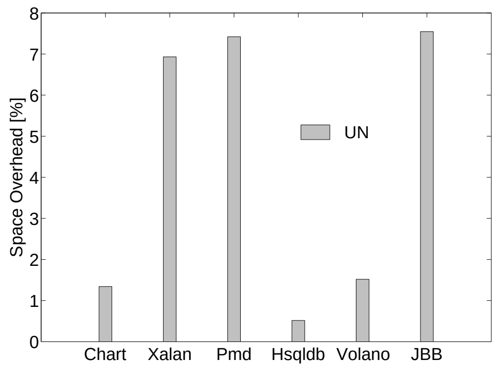


Figure 3.11: Heap space overhead across the heap sizes for concurrent unbounded MC (UN). We use the same heap size ranges as in Figure 3.12.

benchmarks, the space overhead of concurrent UN averages at 4.1%. Note that concurrent UN requires about 28% more heap space than STW UN (Section 3.3.6). Thus, bounding space overhead in concurrent MC does not seem necessary/practical.

Throughput

In Figure 3.12, we present per-benchmark graphs, each of which shows execution time as a function of heap size (starting from the minimum heap size). For the minimum heap sizes, concurrent UN improves throughput by up to 52% (Xalan) and by 29% on average (relative to the concurrent Compressor).

Pause Times

Figures 3.13(a) and 3.13(b) present average and maximum pause times for concurrent UN and concurrent CP. For each benchmark, we report the average value across the

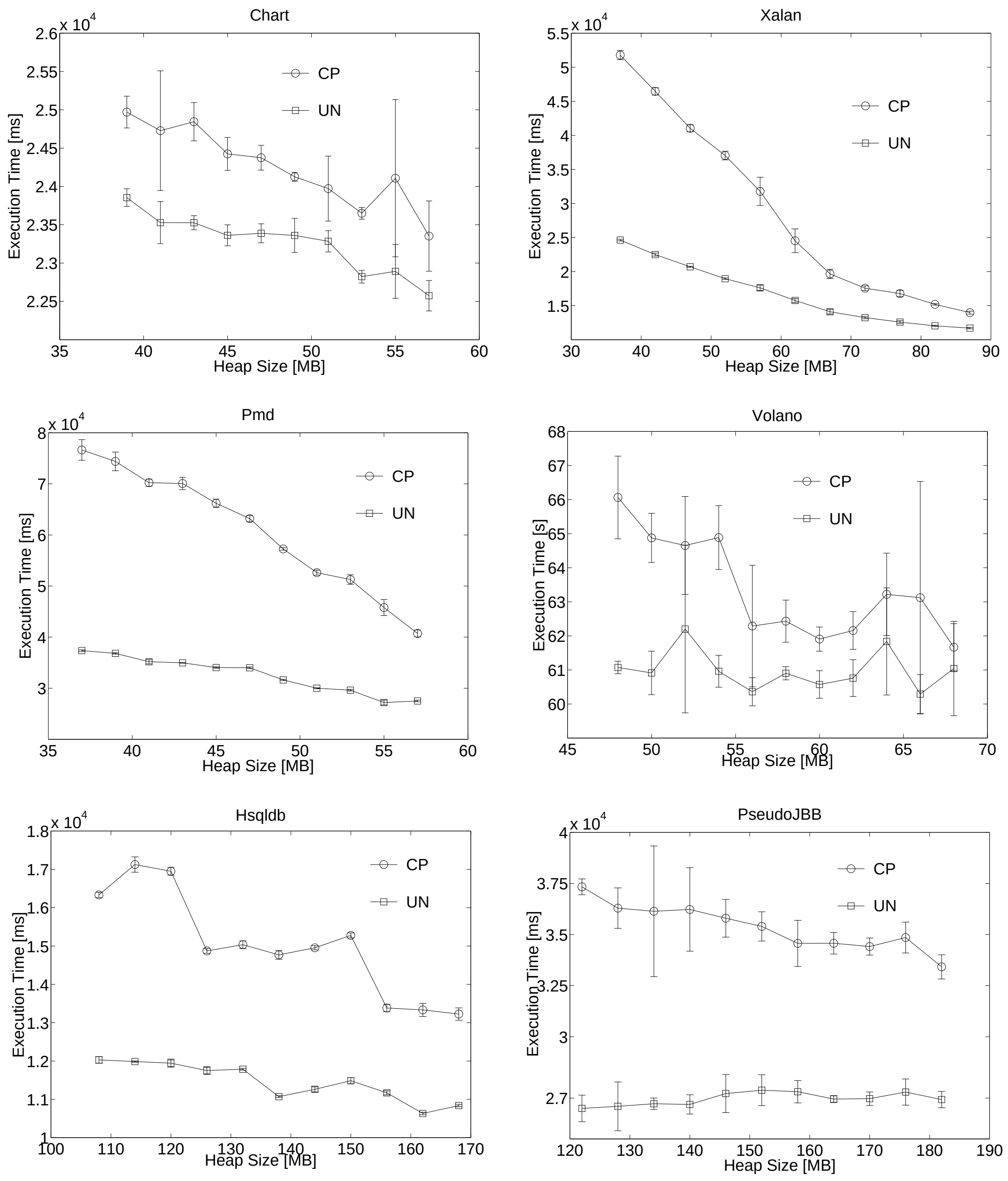
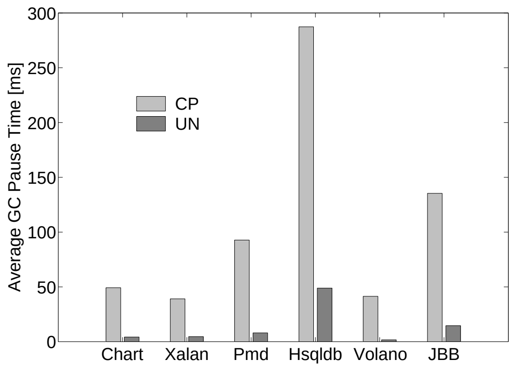


Figure 3.12: Benchmark performance (execution time) across the heap sizes for concurrent unbounded MC (UN) and concurrent Compressor (CP). Error bars indicate the standard deviation across 3 runs.

(a) Average pause times



(b) Maximum pause times

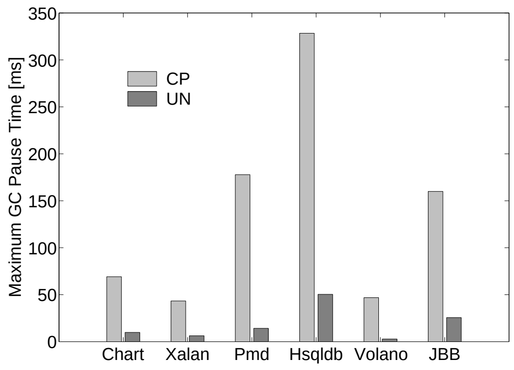


Figure 3.13: GC pause time statistics across the heap sizes for concurrent unbounded MC (UN), and concurrent Compressor (CP): average pause times (a) and maximum pause times (b). We use the same heap size ranges as in Figure 3.12.

heap sizes. Note that we do not consider pauses imposed by concurrent marking here, only those imposed by concurrent compaction. Compared to concurrent CP, concurrent UN reduces average pause times by up to 96% (Volano) and on average by 90%, while reducing maximum pause times by up to 94% (Volano) and on average by 88%. Since concurrent CP moves objects, it needs to update the pointers in the young and permanent generations as part of its STW phase. Concurrent MC does not need to do that and thus its STW pause is much shorter.

3.3.6 Stop-the-World/Concurrent Tradeoffs

To lend insight into the tradeoffs associated with STW and concurrent compaction [30, 57, 99, 127], we compare STW UN with concurrent UN, in terms of through-

Bench-	Min.Heap[MB]		Max.Pause[ms]		Exec.Time[s]	
mark	SMC	CMC	SMC	CMC	SMC	CMC
Chart	27	39	81.4	7.9	23.13	23.85
Xalan	31	37	66.3	6.2	15.64	24.62
Pmd	31	37	160.4	11.2	22.44	37.36
Hsqldb	100	108	n/a	39.7	9.43	12.03
Volano	33	48	n/a	2.1	60.25	61.07
JBB	92	122	147.0	24.1	23.63	26.49

Table 3.3: Comparison of STW unbounded MC (SMC) and concurrent unbounded MC (CMC). We report the minimum heap sizes, maximum GC pause times, and execution times. Execution times and pause times are obtained for the minimum heap size of CMC (as it is larger than the minimum heap size of SMC). The reported pause times correspond to compaction only (marking is excluded).

put, pause times, and memory footprint. Both compactors use the same STW parallel marking algorithm.

Table 3.3 shows experimental results for our benchmarks. We report the minimum heap size in MB (columns 2 and 3), maximum pause time in ms (columns 4 and 5), and execution time in seconds (columns 6 and 7). Execution time and pause times are measured for the minimum heap size of concurrent UN (shown in column 3). This heap size is often much larger than the minimum heap size of STW UN – in some cases big enough to prevent STW UN from any GC activity (we then report pause times as n/a).

Concurrent GC trades pause times for throughput and heap space. On average, relative to STW UN, concurrent UN requires 28% more heap space and degrades throughput by 28%. Maximum pause times (needed for compaction, not marking), however, are shorter for concurrent UN by 89% on average.

Bench-mark	# System Calls	GC Time [ms]	System Time [ms]	System Time [%]
Chart	4358	3405	13.5	0.4
Xalan	92796	8869	287.7	3.2
Pmd	26585	8319	82.4	1.0
Hsqldb	911	6818	2.8	0.0
Volano	12514	19592	38.8	0.2
JBB	299953	41134	929.9	2.3

Table 3.4: The cost of unmapping system calls in STW unbounded MC. We report the total number of the `mmap` calls, total GC time, total time spent in the system calls, and percentage of GC time spent in the system calls. System time has been conservatively estimated using a serial micro-benchmark.

3.3.7 Unmapping Overhead

We have evaluated the cost of the `mmap` system calls relative to GC time in STW UN. Table 3.4 presents per-benchmark data obtained for the minimum heap sizes. We report total number of system calls, total GC time, total system call time, and percentage of GC time spent in system calls. We estimate the cost of a single unmapping system call using a separate micro-benchmark. Our platform needs 3.1s to perform 10^6 unmapping calls. The length of the unmapped region does not impact this cost. On average, STW UN spends 1.2% of GC time in system calls. Note that this result is an upper bound as our micro-benchmark is not parallel.

3.3.8 Other Benchmarks

Thus far, we have only presented detailed experimental data for a subset of benchmarks that we have studied. For our in-depth analysis we have selected standard, benchmarks whose performance is considerably affected by GC. Table 3.5 summarizes the experimental data obtained for the remaining deskside utility benchmarks that we have investigated: Db (memory-resident database) and Javac (Java compiler) from the SPECjvm (1998) suite [145] as well as Bloat (bytecode analyzer/optimizer), Fop (XSL parser and formatter), and Lusearch (text search engine) from the DaCapo (2006) suite [54].

In Table 3.5 we report results for both STW and concurrent UN (slash-separated) in comparison to STW HS and STW/concurrent CP. We report the minimum heap size for STW/concurrent UN (column 2), space overhead for STW/concurrent UN (column 3), and average pause time reduction in comparison to HS and CP (columns 4–5). Column 4 compares STW UN and STW HS. Column 5 compares STW UN with STW CP as well as concurrent UN with concurrent CP.

For our additional benchmarks, on average, concurrent UN requires 36.5% more heap space than STW UN. Space overhead, across these benchmarks, is 6% for STW UN and 3% for concurrent UN. Concurrent UN reduces average pause times by 92% compared to concurrent CP. STW UN reduces average pause times by 59% relative to STW HS and by 70% relative to STW CP.

Bench-mark	Heap Size [MB]	% Space Overhead	% Avg. Pause	
			vs. HS	vs. CP
Bloat	16 / 20	5.9 / 3.1	69.6	73.0 / 96.9
Fop	20 / 24	3.1 / 1.0	57.3	68.8 / 91.6
Lusearch	16 / 24	4.4 / 3.1	62.6	73.7 / 97.2
Db	24 / 32	0.8 / 0.5	47.6	66.7 / 94.2
Javac	24 / 37	15.7 / 8.5	56.1	68.8 / 80.8

Table 3.5: GC statistics for additional benchmarks using the minimum heap sizes. Slash delimits data for STW and concurrent MC. In subsequent columns, we report the minimum heap size for STW/concurrent unbounded MC (2), space overhead for STW/concurrent unbounded MC (3), average pause time reduction for STW unbounded MC relative to STW HS (4) and STW Compressor (5), and average pause time reduction for concurrent unbounded MC relative to concurrent Compressor (5).

3.4 Related Work

While most prior work on parallel/concurrent compaction has focused on virtual-memory-oblivious compactors, the interaction between the collector and virtual memory has recently gained interest [101, 175, 80, 46]. Previously reported compactors achieve compaction by moving all (or some [57, 85, 46]) live objects and need at least two phases. MC attempts to achieve compaction without any object moving and is a nearly-single-phase compactor. Following the methodology used in [101], we define a GC phase as an operation with cost proportional to the heap or live data size. The first phase in state-of-the-art compactors is marking [99] which identifies live objects through parallel/concurrent tracing.

3.4.1 The Compressor

The Compressor [101] is a parallel compacting GC that requires two phases: marking and compaction. It supports both stop-the-world (STW) and concurrent collection. The Compressor (herein referred to as CP) uses virtual memory operations (i.e. page mapping and unmapping) but accomplishes compaction by moving live objects and adjusting the pointers. The compaction is perfect (i.e. heap fragmentation is fully eliminated). The compactor employs two virtual spaces and copies objects page by page from one space to the other. CP, akin to a copying collector, always moves all objects. It updates pointers after moving using information it has recorded in auxiliary data structures (which include the block-offset array). This process is accompanied by freeing pages in the source space and allocating pages in the destination space. CP imposes a small constant space overhead (1.5% for 256-byte blocks) for auxiliary data structures. By contrast, MC performs compaction in nearly one phase and eliminates object moving and pointer adjustment. MC imposes a variable space overhead (on average $<$6%, which can be bounded). Both compactors preserve object order. CP unmaps and maps the entire heap each time the compaction is invoked. MC limits the number of virtual memory operations to the number of dead-object clusters.

3.4.2 The HotSpot Compactor

The parallel compactor currently available in the HotSpot JVM [85] (herein referred to as HS) is similar in spirit to the Compressor. HS updates pointers in the same way but moves objects only when it is necessary. HS is a STW virtual-memory-oblivious collector with two phases: marking and compaction. HS divides the heap into fixed-size regions (chunks) and uses a liveness bitmap to record the locations of live objects. During marking, HS computes additional per-chunk data needed for pointer adjustment. The compaction phase is parallel. Threads claim available regions atomically and fill them with live objects. A region becomes available when all its objects have been evacuated (it is empty) or it has been compacted onto itself. HS updates interior object pointers as it fills regions. Filling a region does not require synchronization and involves identifying source objects destined for the region and copying them until the region is full or no more objects are left. HS computes a new location of a live object as the start of its destination region plus the size of live objects that precede the object in that region. HS performs perfect, sliding compaction and preserves the object order. HS imposes a constant space overhead of 3% (needed for per-region data that includes the current compaction state for each region). The advantages of MC over HS are similar to those over the Compressor: nearly one GC phase (instead of two) and avoidance of object moving and pointer manipulation.

3.4.3 The IBM Compactor

The IBM collector [1] is a parallel STW compactor that comprises three phases: marking, object moving, and pointer fix-up. This collector does not manipulate virtual memory mapping. It does not guarantee perfect compaction and, therefore, imposes an application-specific space overhead, similarly to MC. The system divides the heap into fixed-size blocks. Initially, GC threads perform intra-block compaction and proceed to inter-block compaction as free contiguous areas begin to appear in the already-compacted blocks. In the moving phase, the system collects information needed for pointer adjustment. In the final phase, the system divides the heap into as many areas as there are GC threads, and each thread redirects pointers in its own area. Pointer adjustment is performed in a similar way as in the Compressor. In contrast, MC neither moves objects nor updates pointers and is a nearly-single-phase collector.

3.4.4 The Flood Compactor

The compactor presented by Flood et al. [67] is a parallel version of the Lisp2 [99, 47] collector. This STW GC requires four phases: marking, forwarding pointer installation, pointer adjustment, and object moving. The heap is divided into p contiguous regions where p is the number of parallel GC threads. The sliding direction alternates between left and right for even and odd regions and as a result $\frac{p}{2}$ groups of objects are formed in the heap. Thus, free space is consolidated only partially. This compactor uses forwarding pointers instead of the block-offset array and the mark-bit vector. Thus, pointer updates are more efficient but one additional phase is necessary. MC achieves higher-quality compaction (the free space is mostly consolidated, no object groups are formed) in nearly one phase and without object moving and pointer adjustment.

3.4.5 The Pauseless GC

The Pauseless GC [46] is a parallel and concurrent compactor that avoids STW pauses through hardware read barriers, fast user-mode trap handlers, an additional intermediate TLB privilege level, and fast cooperative preemption via interrupts. The compactor consists of three phases, called mark, relocate, and remap, each of which is parallel and concurrent. The mark phase periodically refreshes the liveness bitmap. The relocate phase uses the most up-to-date liveness bitmap to find pages that contain few live objects, evacuates live data from those pages, and frees the underlying physical memory. Pages with no live data are unmapped as in MC. Evacuated virtual pages containing live objects are protected to trigger traps upon access. The system maintains pointer-forwarding information outside of the evacuated pages, in side arrays (hash table), and imposes variable, but small, space overhead. Mutators using stale pointers raise traps which in turn update pointers to refer to new object locations. The remap phase traverses the object graph executing a read barrier against each pointer to ensure the completeness of lazy pointer forwarding and thus guarantees that all evacuated virtual pages are eventually unmapped. The system performs the remap phase concurrently with the mark phase of the next collection cycle. Unlike the Pauseless GC, MC performs compaction in a nearly one phase – marking, which can be implemented either as stop-the-world or concurrent. MC does not require special hardware support, never copies objects, and reclaims only completely free pages, all of which significantly simplify implementation.

3.4.6 Virtual Memory Support for GC

Recently proposed collectors that leverage virtual memory either focus on copying, not on compaction (like MC), or aim at reducing heap space usage, not at avoiding object moving (like MC). For example, MarkCopy [136] leverages virtual memory mapping to reduce the space overhead of a copying collector. The collector does not require a copy reserve since it maps and unmaps consecutive pages as copying progresses (in a way similar to that of the Compressor [101]). Unlike MC, these approaches involve object moving.

Collectors that cooperate with the virtual memory manager to reduce the collector-induced paging [77, 175, 80, 179, 174] are orthogonal and complementary to MC. The Bookmarking collector [80] records summary information about outgoing pointers from evicted pages to avoid accessing non-resident pages during full-heap compacting collections. CRAMM [175] and IV heap sizing [77] use VM paging behavior to predict and set dynamically an appropriate, application-specific, heap size that adapts to changing memory pressure.

The Boehm-Demers-Weiser [35] garbage collector is a mark-sweep (non-compacting) collector for C/C++ which uses page unmapping as an optional and supplementary mechanism to reduce fragmentation. This collector is conservative (i.e. not all garbage can be identified). Page unmapping in the context of conservative GC for C/C++ has also been investigated in [132]. The proposed collector remaps virtual memory pages to reduce external fragmentation in a free list of large objects. In contrast, MC employs unmapping as a primary technique to achieve compaction and is the first to do so among non-conservative (precise) collectors. Doug Lee's malloc library [103] uses mmap/munmap primitives for memory allocation/reclamation. This system, however, does not support or provide garbage collection.

An alternative to STW collection is concurrent GC, commonly employed for server systems, which interleaves application (mutator) and GC execution via additional synchronization and resource (memory and processor) over-provisioning, to reduce GC pause times. The concurrent version of the Compressor [101], Garbage-First collector [57], and mostly-concurrent mark-sweep [122] are recent examples of concurrent GCs. Concurrent collectors commonly protect virtual pages in order to detect conflicts with mutators and to exploit cache locality [101]. Extant systems supporting concurrent/parallel collection either do not attempt compaction [127, 122, 121, 19, 20, 34] or move/copy live objects [101, 82, 43, 4, 57]. In contrast, MC achieves compaction without object moving.

3.5 Summary and Conclusions

The Mapping Collector (MC) is a generational, parallel GC that supports both stop-the-world and concurrent compaction. MC coordinates with the underlying virtual memory system of the operating system and performs compaction in nearly one phase. Thus, MC is simpler and more efficient than state-of-the-art compactors which require at least two phases. Unlike previously reported compactors, MC is a non-moving collector that leverages the level of indirection provided by virtual memory to consolidate free space into a single contiguous region. By doing so, MC avoids costly object copying and pointer adjustment. The motivation for MC is the observation that unreachable objects in the heap tend to form clusters that can be effectively reclaimed at the granularity of virtual pages. Space overhead imposed by MC is variable but modest in practice and can be bounded by relatively infrequent fall-back to conventional, perfect compaction. MC is particularly attractive for concurrent compaction as it does not require synchronization with the mutators and its space overhead is not a problem in the light of heap over-provisioning.

We implement MC in the open-source HotSpot JVM and evaluate it experimentally on a multiprocessor using a range of different benchmarks and metrics, including throughput, pause times, and scalability. We show that MC significantly outperforms state-of-the-art, stop-the-world parallel compactors (the Compressor and the HotSpot compactor), as well as the concurrent Compressor, for the metrics and benchmarks that we investigate.

The text of this chapter is in part a reprint of the material as it appears in [159].

Chapter 4

Dynamic Prediction of Collection Yield: Improving Intra-Runtime Memory Management Performance Using Virtual Memory

In this chapter, we describe another approach to improving the performance of intra-runtime memory management by using OS support for virtual memory. However, in contrast to Chapter 3, which focuses on leveraging page mapping operations to improve the collector performance, this time we investigate exploiting page reference bits maintained by the OS kernel to avoid unproductive collections. Specifically, we discuss the design and implementation of a collection yield predictor that enables to estimate the amount of heap space that can be reclaimed by a collection. Unlike extant MREs that trigger collection based on heap use, our system schedules collection at times when the yield is sufficient to justify the GC cost. The predictor builds on the observation that unreachable objects in the heap form clusters that span pages that are never referenced by an application. Counting such never-referenced pages proves to be an accurate estimate of what percentage of the heap is currently reclaimable. This allows to avoid low-yield collections and improve performance significantly.

4.1 Introduction and Motivation

To support the vast diversity of deskside and server workloads, modern system software stacks have grown both in depth and complexity and now commonly include managed runtime environments (MREs), e.g. Java and C# virtual machines, layered on top of a general-purpose operating system (OS), e.g. Linux. Although independent and isolated, the OS and MRE layers provide similar services for programs, such as memory management and access to protected resources. In this chapter, we investigate how to better coordinate the activities of memory management between the hardware, OS, and MREs to improve the performance of applications.

Garbage collection (GC), commonly employed by MREs to increase programmer productivity and software reliability, can negatively impact both application throughput and interactivity. Key advances in GC that have led to significant reduction of collection costs include support for parallelism and concurrency, generational heap layout, and compaction [99, 85, 101, 46, 47, 159, 57, 122]. Moreover, recent GC systems introduce ways to better coordinate the activities of the MRE GC and OS virtual memory subsystem [159, 101, 77, 136, 175, 80, 179, 174, 79].

We build upon and extend this prior work to improve MRE-OS-hardware interaction in a way that facilitates *collection avoidance*. In particular, we design and implement a lightweight prediction scheme (the Yield Predictor) that identifies, with low overhead, the amount of free space a particular GC invocation is likely to yield from dead objects. GC systems can employ this yield prediction to avoid ineffective collections that are unable to reclaim sufficient space to justify the incurred cost, by trading off a small space overhead (equal to the small yield that would have been collected by the skipped GC). Most extant systems trigger GC unconditionally when a program exceeds some threshold on its heap use, without regard for GC yield. Systems that trigger GC proactively, rely on complex monitoring and analysis of program behavior [173].

The Yield Predictor (YP) provides a simple solution to distinguishing productive GCs by estimating GC yield using hardware page reference bits that the OS uses to implement virtual page replacement. Key to its efficacy is the statistical property of modern programs that dead objects tend to cluster together in large groups (larger than the 4KB virtual page size), and that pages that have not been recently referenced by the application correlate well with dead clusters. We validate these properties with empirical data and describe how YP makes use of them to estimate GC yield. We implement YP for three state-of-the-art parallel compactors within the production-quality, open-source, HotSpot Java virtual machine. These compactors represent three distinct canonical heap layouts and GC strategies (compaction by sliding, copying, and remapping).

YP demonstrates that MREs can significantly benefit from existing architectural support for memory management when GC is given access to page tables and can leverage the standard hardware mechanism used for marking pages referenced by a process. Thus, the predictor identifies a new use for the existing hardware facility. In addition, YP enables a more resource-efficient GC mechanism that gives the memory manager more control over the space/time trade-offs and allows for well-informed GC scheduling decisions at run-time.

A comprehensive experimental evaluation of YP based on standard community benchmarks and open-source applications shows that YP consistently provides high prediction accuracy and that avoidance of unproductive GCs can substantially improve (44–59%) the performance of both server- and client-side benchmarks.

In the following sections, we describe the design and implementation of YP (Section 4.2), present results of our empirical evaluation (Section 4.3), discuss related work (Section 4.4), and conclude (Section 4.5).

4.2 Design and Implementation

State-of-the-art MRE memory management systems trigger GC upon complete (or partial, in the case of concurrent GC) exhaustion of the space designated for object allocation. The key limitation of this approach is that every collection is performed irrespective of whether it is worth paying for. Our investigation of *GC productivity*, i.e. the total size of dead objects that a collection cycle reclaims, shows that many GCs are unproductive and are able to reclaim only a small fraction of the heap. We empirically evaluate this phenomenon further in Section 4.3.3. We observe from our experiments that a number of different (deskside and server) Java applications have an average GC yield below 5% of the heap space.

If we are to skip unproductive GCs, we must have a fast and accurate mechanism for estimating the total size of dead objects in the heap. Marking, which identifies live data via traversing the reachable object graph starting from the roots, is the most commonly-used mechanism to do this and can precisely compute the amount of dead space. However, marking takes significant time and, according to our measurements, comprises between 50% (for the Compressor) and 90% (for the Mapping Collector) of total GC time, depending on the compaction algorithm. Thus, partial reduction of collection cost by skipping only the phases that follow marking is not satisfactory due to its lower potential for performance improvement.

Our goal is to design and implement effective, practical, and lightweight yield prediction. Our approach to enabling such prediction relies on improved coordination between the activities of OS, hardware virtual memory subsystem, and MRE memory management.

4.2.1 Yield Predictor Design

General-purpose OSes support virtual memory to isolate address spaces of distinct processes and provide a convenient uniform linear addressing. Most virtual memory implementations divide the virtual address space of a process into *pages*, each typically 4KB in size. The mapping between virtual pages and physical page frames is stored in an OS-maintained *page table*. Under memory pressure, the kernel uses *swap space* to evict pages that are unlikely to be accessed in the future. To implement swapping, pages tables reserve two bits per page, indicating whether a specific page is *dirty* and has been recently *referenced*. These bits are set by hardware upon memory store/read.

Prior work shows that for modern Java applications, objects with similar life spans tend to be spatially clustered in the heap and that dead objects often form clusters larger than the 4KB virtual page size [159, 167]. The design of YP leverages these statistical properties and the observation that dead pages are never accessed by the program and, as a result, eventually become not-recently-referenced (NRR) from the OS kernel perspective. YP exploits this relationship between NRR and dead pages to estimate the total number of dead pages in the heap. When the number of dead pages is small, an impending garbage collection is likely to be ineffective, i.e. unable to reclaim sufficient space to justify the cost of performing GC. We employ YP to avoid such collections in state-of-the-art compaction systems. We trade a small space overhead for significant performance gains that result from skipping low-yield GCs. We analyze these trade-offs in Section 4.3.

YP takes two parameters: *skip threshold* and *young-old ratio*. We investigate YP's sensitivity to both in Section 4.3. The skip threshold determines the free proportion of the heap that is necessary to trigger regular collection (e.g. for the skip threshold of x we skip all GCs that we predict to reclaim not more than x% of heap space). The young-old ratio identifies pages that are recently-referenced (RR). YP considers a time window between two consecutive GCs and divides it into two contiguous parts: young and old, according to the young-old ratio. Pages with a last-access timestamp in the young partition are considered to be live.

There are two sources of potential inaccuracy in YP's yield prediction process. The first is a page that we identify as NRR, that is not actually dead but is instead, not accessed recently. The second is a page that is dead that we have not yet identified as NRR. Although all dead pages are guaranteed to be found eventually, there may be a delay before YP correctly classifies a page as dead.

Since YP considers only the total number of dead pages in the heap, as opposed to determining the status (live/dead) for each individual page, the dead space estimation errors that these two phenomena introduce do not add up, but instead, cancel each other out. Thus, the goal of YP parameter tuning is to make sure that the two misclassifications (dead-as-live and live-as-dead) occur at similar frequency. Therefore, to optimize accuracy we need to choose the best old-young ratio. For large ratios live-as-dead misclassification dominates. For small ratios dead-as-live misclassification dominates. For the right choice of the young-old ratio the two misclassifications are similarly frequent and YP accuracy reaches its optimum.

YP periodically consults the OS kernel to obtain a list of recently-referenced (RR) pages within the heap. A dedicated polling thread in the MRE wakes up at regular time intervals and retrieves the addresses of RR pages. For each page in the heap, YP records the time when a page was last believed to be RR, using a `timestamp` array stored in the MRE.

Each time the polling thread tests the reference bits, it clears them atomically. To avoid interference with the kernel swapping mechanism that also relies on RR bits, we introduce two new bits per page: *mre-cleared* and *os-cleared*. This extension is software-only. YP shares hardware RR bits with an OS, but whenever YP (or an OS) clears a hardware RR bit, we set the mre-cleared (or the os-cleared) bit in software so that no information is ever lost. We multiplex hardware bits and use software bits to indicate if a HW bit was cleared. To read an RR bit of a page, YP (or an OS) computes a *logical-or* of the hardware RR bit and the os-cleared bit (or the mre-cleared bit). When MRE (or an OS) clears the RR bit, it also must clear the os-cleared (or mre-cleared) bit.

Key to our approach is accurate RR page tracking. First, we must distinguish between an application access to a page and a GC access to a page, and only consider the former as an RR trigger (since GC may reference pages not reachable by the program). To enable this, before every minor/major collection, YP takes a snapshot of the current RR page bits, and after each GC clears the reference bits that are set as a side-effect of the collection. Moreover, we disable the polling thread during GC.

In addition, YP measures the time spent in GC and advances the values in the `timestamp` array accordingly after each collection. This is necessary to eliminate the impact of the stop-the-world GC pauses on timestamps. During GC pauses, mutators are inactive and live pages are not used, which can make them appear to be NRR. Advancing timestamps eliminates this problem.

YP maintains a boolean array (`mispredicted_dead`), for all pages in the heap. Each entry indicates whether a live page has been misclassified as dead. Such pages are never considered dead again. The intuition behind this is that many applications allocate permanent data structures that subsequently are rarely used which can lead to YP false positives.

Since compactors typically move objects (MC is the exception), after each collection for such compactors, YP recomputes the `mispredicted_dead` array to reflect the relocation that has occurred in the heap. For each live target page, we consider the (live) source pages containing objects being moved to the target page. If any source page has ever been misclassified as dead (including during the current GC cycle) then the target page is marked as `mispredicted_dead`. We perform this propagation since the target page is also likely to be misclassified.

YP makes predictions of the potential GC yield while mutators are suspended (i.e. at a safepoint). Prediction is short and simple and therefore does not need to execute concurrently. Parallelization is not necessary either as prediction cost is proportional to the number of pages in the heap. The polling thread executes concurrently and asynchronously to mutators. It does not employ locking or synchronization and imposes negligible overhead, especially when executed on a separate CPU/core.

4.2.2 Yield Prediction Process

Table 4.1 shows the pseudocode for the steps that YP executes during each full GC. YP first stops the polling thread. Next, it obtains RR pages and updates the page timestamps (lines 1–4). The predictor then iterates over the heap pages to determine which pages are dead (lines 5–14); YP skips any pages previously found to result in false positives (`mispredicted_dead[page]` is true). For other pages, we consider

```
rr_list = get_rr_pages(heap_start, heap_end)
for page in rr_list do
    timestamp[page] = current_time
end for
dead_cnt = 0
limit = OLD_YOUNG_RATIO $\cdot$ (current_time $-$ last_full_gc)
set_all_entries(predicted_dead, false)
for page in [heap_start, heap_end] do
    age = current_time $-$ timestamp[page]
    if age $>=$ limit and not mispredicted_dead[page] then
        predicted_dead[page] = true
        dead_cnt += page_size
    end if
end for
if dead_cnt $\leq$ SKIP_THRESHOLD $\cdot$ heap_size then
    dead_cnt = max(dead_cnt, min_expansion)
    expand_heap(dead_cnt)
    total_expansion += dead_cnt
else
    fall_back_to_regular_gc
    try_to_shrink_heap(total_expansion)
    update_if_heap_shrunk(total_expansion)
    set_all_entries(propagated_dead, false)
    for page in [heap_start, heap_end] do
        if has_live_objects(page) then
            if mispredicted_dead[page] or predicted_dead[page] then
                target = relocation_target(page)
                propagated_dead[target] = true
                if crosses_next_page(page, target) then
                    propagated_dead[successor(target)] = true
                end if
            end if
        end if
    end for
    mispredicted_dead = propagated_dead
    update_if_relocated(heap_start, heap_end)
end if
clear_rr_pages(heap_start, heap_end)
for page in [heap_start, heap_end] do
    timestamp[page] += gc_time
end for
last_full_gc = current_time
```

Table 4.1: Pseudocode for yield prediction executed by YP during each full collection. SKIP_THRESHOLD and YOUNG_OLD_RATIO are the two YP parameters.

their age, i.e. how long since they were accessed, and compare that against a percentage of the distance (in time) between the current and previous full GC. This percentage is a YP parameter (the young-old ratio).

If the predicted amount of free space is larger than the skip threshold, the system falls back to regular compaction (i.e. does not skip GC). Following compaction, YP attempts to shrink the heap back to the size it was prior to GC-skipping (if any) to reduce space overhead (line 21). Finally, we re-compute the `mispredicted_dead` array (lines 23–35), using the auxiliary array `propagated_dead`. We traverse the live heap pages computing a target location for each such page. If the source page has ever been mispredicted dead or has been predicted dead in the current GC cycle (note that this page is live), the target page becomes `mispredicted_dead`.

If the predictor expects low yield, it skips the compaction and grows the heap (lines 16–18). The expansion corresponds to the predicted free space, but is never smaller than the minimum value (`min_expansion`, 128KB in our implementation). This minimum is necessary to ensure mutator progress, i.e. to ensure that the mutator is able to allocate the data that triggered the GC originally. In the GC epilogue (lines 38–42), we clear the reference bits and advance the timestamps by the GC pause time.

We never skip the first collection as it is typically highly productive. Instead, we use this collection to bootstrap the predictor and initialize its data structures.

When skipping an unproductive collection, we extend the heap by the estimated total size of dead objects (which themselves are not reclaimed as without marking one cannot identify them). This creates space overhead. This overhead is small in practice because we skip only low-yield collections and shrink the heap when possible on subsequent full collections.

Note that GC skipping is substantially different than heap over-provisioning. Executing an application with a larger heap does not prevent unproductive GCs, although it does reduce the total number of collections. YP ensures with high probability that the system triggers GC only when it is worth doing so. Thus, YP enables better resource management and gives the memory manager greater control over space/time trade-offs. For example, when an expensive GC algorithm is used, an MRE might be more conservative when deciding to trigger a collection. In addition, the user need not determine the right heap size a priori.

Note that on 64-bit platforms, the space costs are the same as on 32-bit platforms – the arrays that we use have one entry per page and pertain only to the area used and mapped by the old generation.

With concurrent GC, YP has similar or even more potential for improving performance. Each cycle of concurrent GC, despite imposing shorter pause times, costs more than the corresponding cycle of the stop-the-world GC.

4.2.3 Implementation Details

We integrate YP into three state-of-the-art parallel compactors in order to investigate its generality and applicability to different collectors. These compactors represent three canonical heap layouts that underlie all modern GC algorithms (including concurrent ones). We use exactly the same prediction algorithm with each compactor.

We implement YP in HotSpot [118], an open-source (GPL), production quality JVM written in C/C++ (source code version 7-ea-b10, released 3/2007). HotSpot uses a generational [155] heap layout comprising the permanent, old, and young generation. The permanent generation contains run-time meta-data for the loaded classes. The young generation is further subdivided into *eden* and two equally-sized survivor spaces (called *from* and *to*). Objects are initially allocated in the eden. Within the young generation a copying collector [67] evacuates live objects from the eden-space and from-space to the to-space and promotes objects that survive several minor collections to the old generation. Major collection (compaction) takes place upon space exhaustion in the old generation.

4.2.4 Kernel Extensions

We have implemented YP using Linux kernel 2.6.17 configured with high memory disabled and SMP enabled. YP consists of a kernel module which, upon loading, creates a new entry in the *proc* filesystem using the `proc_mkdir` and `create-`

`_proc_info_entry` functions. The entry is located at */proc/ref/bits* and is writable but not readable. A polling thread in an MRE repeatedly sleeps for 10ms, opens the */proc/ref/bits* file, writes three words to it (in order to obtain a list of RR pages within a given address range), and closes the file. These words are: start and end addresses of the memory range plus a pointer to an array for the results. The kernel invokes the callback registered by the module, copies the three words from the user space, inspects page table entries corresponding to the specified address range and copies the results into the MRE-provided array (in userland). The first array entry contains the number of the returned pointers to RR pages.

We obtain the page table entries (PTEs) for subsequent pages using the macros: `pgd_offset`, `pud_offset`, `pmd_offset`, and `pte_offset_map`. We clear the reference bits in PTEs atomically after testing with the help of `ptep_test_and-_clear_young`. The polling thread holds a spin lock for the page table of the current process during the entire operation.

To avoid interference with kernel swapping, we make use of the two unused bits in page flags (bits 21 and 22) which we define as *PG_kernel_cleared* and *PG_mre_cleared*. Each physical page managed by the kernel has a page frame descriptor (*struct page*) associated with it, which contains an *unsigned long flags* field. The flags determine if a page is referenced, dirty, locked, etc. Note that these software flags are distinct from hardware page flags present in PTEs. The kernel module sets the *PG_mre_cleared* bit whenever clearing the RR bit in a PTE. The kernel sets the *PG_kernel_cleared* bit whenever it clears the RR bit in a PTE. The latter requires minor modification to the kernel (the code fragments that get/set PTE RR bits).

4.2.5 Alternative Approaches

We have investigated two other approaches to implementing YP: mlock-based and kswapd-based. The mlock-based design employs page pinning for the old generation (via the POSIX `mlock` system call). Pinning eliminates interference of page access bit clearing (done by an MRE) with kernel swapping mechanism. This approach is simple but requires heap pages to be locked in physical memory.

In the kswapd-based design, instead of an MRE periodically clearing page access bits, we reuse an existing kernel thread (kswapd daemon) and decrease its sleeping interval. Kswapd clears page access bits whenever it wakes up. We increase the frequency of kswapd wake-up to match the bit clearing frequency needed by YP. MREs have read-only access to RR bits and the kernel swapping mechanism benefits from higher sampling frequency of RR pages (better accuracy). However, this approach assumes the existence of kswapd and its certain behavior (periodic wakeup and RR bits clearing) which makes it less portable (e.g. it works in Linux 2.4 but not in 2.6).

4.3 Experimental Evaluation

We empirically evaluate YP using three state-of-the-art parallel compactors: the Mapping Collector (MC), the Compressor (CP), and the HotSpot compactor (HS). HS and CP are described in detail in Section 3.4 and MC is the subject of Chapter 3.

We first overview our experimental methodology and benchmark suite. Next, we present results from our experiments that measure YP prediction accuracy and cost as well as the impact of YP on the application throughput, GC pause times, and memory footprint. In addition, we systematically evaluate YP sensitivity to different values of its two parameters: *skip threshold* and *young-old ratio*.

4.3.1 Methodology

Our experimental platform is a dedicated dual-core Intel Core 2 Duo (Conroe B2) machine clocked at 2.66GHz with the unified 4M 16-way L2 cache and 32K 8-way L1 cache, 2GB main memory, running Debian GNU/Linux 3.0 configured with the 2.6.17 kernel. The virtual page size is 4KB. We use HotSpot version 7-ea-b10 deployed within OpenJDK 1.6.0 and compiled with GCC 3.2.3, in the optimized client-compiler (C1) mode.

We employ YP for old-generation collection, i.e. full-heap, major GCs, only. Minor collections use a parallel copying collector in the young generation. For each benchmark, we investigate four heap sizes within a range that captures significant to medium GC activity, wherever possible. Each of our experiments uses a fixed-size heap, which consists of the young, old, and permanent generation. The young generation is 25% of the old generation. The permanent generation is 12MB (HotSpot default). We disable all explicit GC invocations and adaptive generation resizing. We employ 2 parallel GC threads as we use a dual-core machine. Survivor spaces occupy 33% of the young generation (the remainder is used by the eden). When reporting heap size, we sum up the size of all three generations.

We repeat each measurement 5 times and report the average as well as standard deviation where appropriate. We evaluate YP in detail for the skip threshold set to 5% and the young-old ratio set to 1%. In addition, we investigate its sensitivity to other skip thresholds (0%, 3%, and 10%) and young-old ratios (2–90%).

Our evaluation is based on 16 Java programs which include standard Java benchmarks and open-source Java applications [76]. We use the subset of the DaCapo [54] and SPEC JVM'98 [145] benchmark suites. In addition, we employ SPEC PseudoJBB'00 [145] and VolanoMark [156]. We selected these benchmarks to capture a wide range of application behaviors while focusing on programs with significant GC activity.

Table 4.2 reports performance data for these benchmarks obtained using HS: heap size ranges (we use 4 heap sizes across each range; heap size includes all generations), execution times, and general GC statistics (for minimum heap sizes): total GC time,

	Benchmark Program	Heap Size Range [MB]	Execution Time [s]	GC Time [s]	GC Count	GC Cost [%]	Reclaimable Space [%]
GPL	beautyj	61 – 64	18.2	13.0	60	71.4	2.0
	findbugs	82 – 97	13.0	2.7	5	21.1	31.0
	jaranalyzer	14 – 17	4.5	0.1	3	3.0	23.2
	javaguard	16 – 22	7.0	3.7	69	53.8	0.5
	jdepend	30 – 33	20.5	6.7	77	32.7	0.3
DaCapo	chart	45 – 48	6.2	0.4	3	5.8	53.7
	fop	14 – 20	4.3	1.9	31	43.6	4.9
	hsqldb	92 – 95	12.2	7.2	11	58.7	0.9
	pmd	40 – 46	6.2	1.2	7	18.6	51.1
	xalan	44 – 68	6.0	1.1	21	18.0	68.7
JVM	compress	41 – 47	2.6	0.0	3	1.8	49.3
	javac	33 – 42	2.9	0.2	3	8.0	60.9
	mtrt	19 – 22	8.6	4.9	97	57.3	0.7
	raytrace	14 – 17	2.7	1.7	58	63.0	0.1
	volano	31 – 34	46.6	16.2	233	34.8	0.3
	psjbb	119 – 125	25.4	12.5	70	49.3	3.3

Table 4.2: Baseline benchmark statistics obtained using HS with YP disabled. Col. 4 is total GC time. Col. 6 is the percentage of execution time consumed by GC. Col. 7 is the average GC yield across heap sizes as the percentage of the old generation size. Highlighted entries are multi-threaded server programs.

total number of GCs, percentage of execution time that is consumed by GC, and average GC yield across heap sizes (as a percentage of the old generation size).

We investigate server-side multi-threaded workloads using VolanoMark (multi-user chat server), PseudoJBB’00 (three-tier database system emulator), and Hsqldb (in-memory database). The remaining benchmarks are deskside utilities: beautyj (source code beautifier), findbugs (Java bug detector), jaranalyzer (jar dependency manager), javaguard (Java bytecode obfuscator), jdepend (dependency analyzer), chart (line graph plotter), fop (XSL-FO parser/formatter), pmd (source code analyzer), xalan (XML

Bench-mark	Average size [byte]			Minimum size [byte]			Maximum size [byte]		
	CP	HS	MC	CP	HS	MC	CP	HS	MC
beautyj	2.0K	2.2K	468.9	24.9	243.3	16.0	137.0K	119.6K	403.1K
findbugs	4.4K	4.4K	4.9K	16.0	16.0	16.0	1.5M	1.7M	2.0M
jaranalyzer	1.6K	1.7K	1.8K	16.0	16.0	16.0	8.8K	28.8K	13.0K
javaguard	430.1	557.9	384.1	28.1	26.4	16.0	7.7K	9.0K	44.0K
jdepend	1.7K	32.5K	245.3	1.3K	2.0K	16.0	7.9K	311.3K	66.7K
chart	77.6K	72.6K	45.7K	15.3	14.7	16.0	2.9M	3.6M	6.5M
fop	1.4K	1.8K	603.0	91.9	88.2	16.0	56.2K	92.5K	147.8K
hsqldb	5.0K	119.5	109.1	16.2	15.8	16.0	1.4M	27.7K	193.5K
pmd	126.6K	187.4K	17.3K	14.8	10.3	11.0	4.5M	6.6M	5.1M
xalan	46.8K	46.4K	127.9K	15.7	15.9	14.5	3.4M	4.0M	14.3M
compress	5.9M	2.9M	5.1M	19.2	25.0	16.0	10.5M	9.2M	13.2M
javac	13.2K	12.9K	8.6K	16.0	16.0	16.0	2.8M	3.1M	4.9M
mtrt	1.9K	2.3K	206.0	120.5	506.5	16.0	186.8K	69.7K	92.9K
raytrace	114.9	579.8	90.8	26.8	544.5	16.0	986.2	808.2	9.5K
volano	486.8	482.7	157.3	48.5	102.7	16.0	8.0K	7.3K	44.1K
psjbb	3.1K	3.0K	997.0	95.5	89.0	16.0	204.5K	329.1K	1.3M
average	397.4K	208.4K	341.0K	116.6	238.8	15.6	1.7M	1.8M	3.0M

Table 4.3: Dead space clustering statistics. For each benchmark, we report average/minimum/maximum dead cluster size across the heap sizes.

to HTML transformer), compress (LZW packer), javac (Java compiler), mtrt (multi-threaded ray-tracer), and raytrace (3D scene renderer).

We run the default variants of the DaCapo benchmarks and use the input size of at least 100 for JVM'98. We execute VolanoMark with 42 chat rooms for 100 iterations and PseudoJBB with 5 warehouses for 10^5 iterations.

4.3.2 Dead Object Clustering

The prediction capabilities of YP depend on dead object clustering, a widely-known phenomenon, previously reported in [159, 167]. We have gathered basic clustering statistics across the benchmarks, such as average, minimum, and maximum cluster

Bench-mark	4KB page coverage [%]			Max. Pred. Cost [%]		
	CP	HS	MC	CP	HS	MC
beautyj	44.1	43.2	81.3	0.0	6.0	5.2
findbugs	92.5	93.3	93.7	3.8	2.8	4.4
jaranalyzer	16.7	22.1	31.4	1.2	1.6	1.8
javaguard	16.0	16.1	51.6	2.0	2.8	1.7
jdepend	18.2	24.4	27.3	0.9	1.3	1.3
chart	99.4	99.4	99.4	2.8	2.3	2.0
fop	26.0	29.0	58.3	2.4	1.5	1.8
hsqldb	22.5	22.6	75.7	9.5	9.5	4.2
pmd	99.3	99.5	98.7	9.1	6.5	5.2
xalan	99.5	99.4	99.8	3.2	4.0	6.3
compress	100.0	100.0	100.0	2.7	1.3	1.9
javac	94.3	95.7	94.7	3.6	3.5	5.3
mtrt	11.6	6.8	31.1	2.1	-0.4	3.8
raytrace	0.4	0.4	13.2	1.2	1.5	2.2
volano	12.4	12.6	35.5	3.6	3.3	3.7
psjbb	41.0	46.2	59.5	8.4	7.9	8.1
average	49.6	50.7	65.7	3.5	3.5	3.7

Table 4.4: YP statistics across the heap sizes: percentage of dead space fully covered by 4KB pages and maximum YP execution time overhead.

size as well as the percentage of dead space fully covered by 4KB pages. The results are summarized in Table 4.3 and in Table 4.4 (Columns 2–4). For each benchmark, we report the average values obtained across all GCs that occurred for the heap size ranges that we use. We have observed that most clusters are smaller than 4KB, however average cluster size is above 200KB. We have found that at least 50% of the dead space is fully covered by 4KB pages. Such clustering generally holds for both client- and server-side Java applications and is stable across inputs and heap sizes.

4.3.3 Collection Yield

Column 7 in Table 4.2 shows the average GC yield for each benchmark. This value is the percentage of the old generation that is reclaimable (averaged across the heap sizes). It indicates how effective the GC is on average at reclaiming dead space. 9 benchmarks have unproductive GCs (yield below 5%). In the remaining 7 benchmarks, the GCs are mostly productive (yield above 23%). We have also observed that the first full collection for all programs is typically productive even if a particular benchmark has a low GC yield on average.

4.3.4 Prediction Accuracy and Cost

We evaluate the prediction error of YP relative to the total heap size as well as relative to the old generation size. Specifically, if the exact amount of reclaimable space is x bytes and the predictor estimates that as y bytes, we compute the prediction error as $|x - y|/size$, where $size$ is heap or generation size. We measure relative error (as opposed to absolute error) because GC yield itself is typically expressed and used in practice as a percentage.

We summarize the accuracy results in Table 4.5, which contains data averaged across the heap sizes. For each benchmark, we report prediction error for the 0% and 5% skip threshold, relative to the old generation size and heap size. The young-old ratio is 1%. The results for the 0% threshold (when no GC is skipped) lend insight

Baseline	Old Generation Size						Heap Size					
Skip Threshold	0%			5%			0%			5%		
Benchmark	CP	HS	MC	CP	HS	MC	CP	HS	MC	CP	HS	MC
beautyj	1.6	1.0	1.0	7.1	7.3	6.2	1.0	0.6	0.6	4.4	4.6	3.9
findbugs	7.1	5.4	5.8	6.7	5.5	5.9	4.6	3.2	3.9	4.3	3.3	4.0
jaranalyzer	9.1	8.8	11.5	9.3	8.5	11.5	1.8	1.8	3.0	1.9	1.7	3.0
javaguard	1.8	2.2	2.1	7.4	7.8	8.3	0.5	0.6	0.6	1.9	2.0	2.2
jdepend	0.6	0.3	0.3	6.6	6.6	6.7	0.3	0.2	0.2	3.2	3.2	3.3
chart	8.6	8.1	7.8	9.3	8.2	7.7	4.7	4.0	4.3	5.2	4.1	4.3
fop	1.7	1.4	1.2	7.2	7.1	7.4	0.4	0.4	0.3	1.7	1.8	1.9
hsqldb	0.4	0.4	0.4	8.7	5.8	7.0	0.3	0.3	0.3	5.8	3.9	4.7
pmd	5.1	12.4	7.8	5.6	11.8	7.0	2.8	6.7	4.5	3.0	6.4	4.1
xalan	2.7	5.7	3.0	2.7	5.5	3.3	1.5	3.3	1.8	1.6	3.2	2.0
compress	4.8	7.4	4.2	4.4	7.5	4.3	2.6	4.0	2.2	2.4	4.1	2.3
javac	4.1	7.0	4.6	3.9	7.0	5.5	2.1	3.5	2.6	2.0	3.5	3.1
mtrt	0.2	0.2	0.3	7.3	8.8	7.9	0.1	0.1	0.1	2.2	2.7	2.4
raytrace	0.1	0.1	0.3	7.1	15.8	7.6	0.0	0.0	0.1	1.5	3.2	1.6
volano	0.6	0.3	0.8	4.8	7.5	9.3	0.3	0.1	0.4	2.3	3.5	4.5
psjbb	2.8	2.8	4.0	6.3	5.9	7.3	1.9	1.9	2.8	4.3	4.1	5.2
average	3.2	4.0	3.4	6.5	7.9	7.1	1.6	1.9	1.7	3.0	3.5	3.3

Table 4.5: Average yield prediction error, across the heap sizes, relative to the old generation size (Cols. 2–7) and heap size (Cols. 8–13). The young-old ratio set to 1%.

into prediction accuracy unaffected by avoided GCs which is important in benchmarks whose GCs are mostly productive.

Across the benchmarks and compactors, average error is below 4% (for the 0% threshold) and below 8% (for the 5% threshold) relative to the old generation size. This corresponds to 2% and 4% relative to the heap size. We investigate accuracy for other thresholds in Section 4.3.6. Accuracy is worse for the 5% threshold because GC skipping increases fragmentation in the heap.

Figure 4.1 shows detailed accuracy plots, across the heap sizes, for selected benchmarks and the 5% threshold. We report average prediction error (data points) and standard deviation (error bars) from 5 measurements relative to the old generation size. The graphs show that accuracy also varies across the heap sizes.

To implement yield prediction, we employ a polling thread in the MRE that periodically samples the hardware page protection bits through an OS kernel module. During each GC, the MRE also executes the YP algorithm (cf. Section 4.2). Both of these operations can impose a performance penalty. The final three columns in Table 4.4 compare the execution times with and without prediction for CP, HS, and MC, to evaluate this overhead. With prediction on, we set the skip threshold to 0%. Thus, we do not skip any collections, and we isolate the performance penalty incurred by YP. That is, for each GC, we do complete prediction and collection work in addition to the polling thread running concurrently. We report the maximum overhead as the percent increase in total execution time, across the heap sizes for CP, HS, and MC. This overhead is below 4% on average. Server-side benchmarks (e.g. hsqldb and psjbb) have the highest overhead as they fully utilize both CPU cores and the polling thread needs to preempt the application threads.

4.3.5 Impact on Applications

In this subsection, we focus on the eight benchmarks with low GC yields, i.e. those below 5% in Column 7 in Table 4.2. In the remaining programs, most collections cannot be skipped (as they are productive) and YP affects performance only marginally (max-

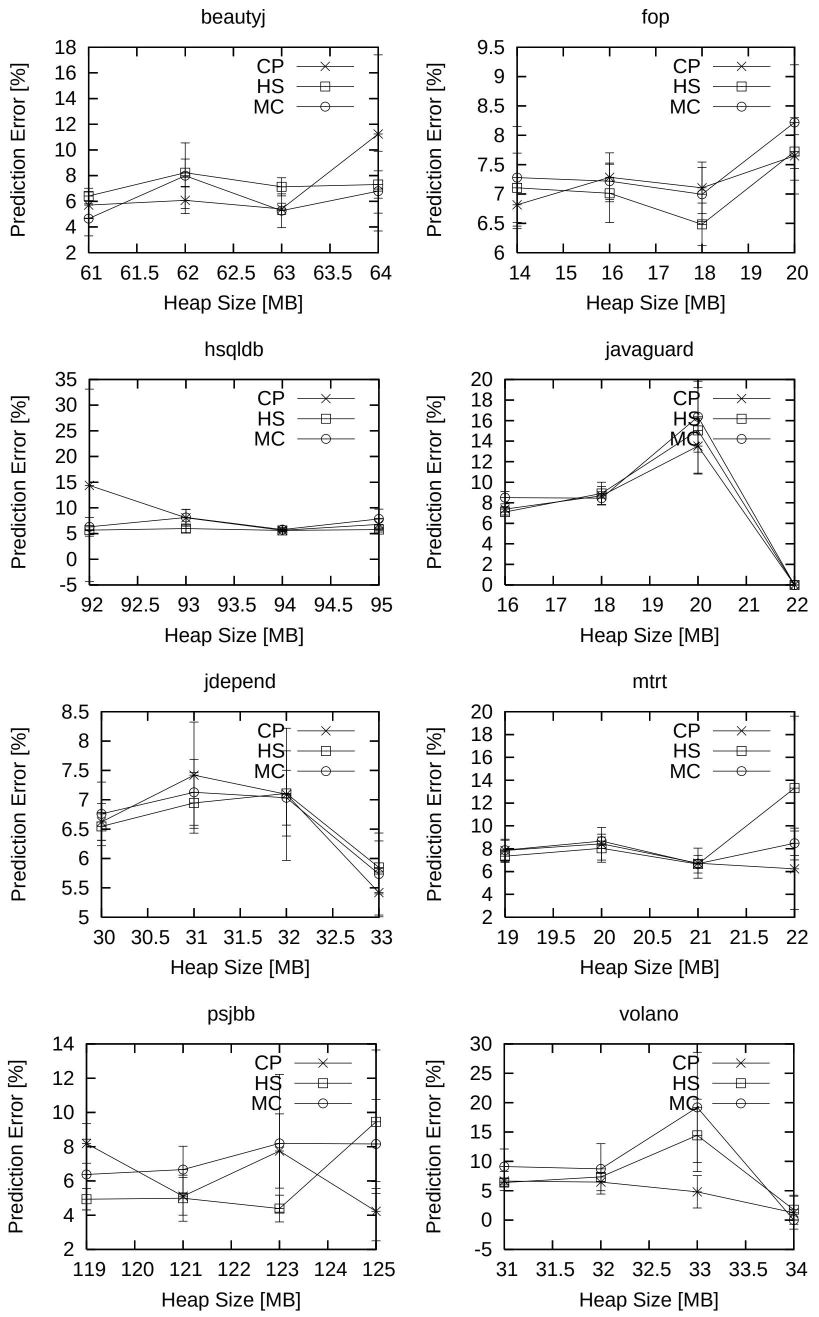


Figure 4.1: Prediction error relative to the old generation size across heap sizes for all compactors and 8 benchmarks (those with the most unproductive GCs). We report average and standard deviation (error bars) from 5 runs. Yield prediction is turned on, the GC skip threshold is 5%, and the young-old ratio is 1%.

Benchmark	Execution Time Reduction [%]			GC Skip Rate [%]			Maximum Pause Time Reduction [%]		
	CP	HS	MC	CP	HS	MC	CP	HS	MC
beautyj	84.8	79.3	74.1	82.9	81.0	81.0	-3.1	1.1	1.3
javaguard	47.5	42.0	31.7	66.7	69.1	69.7	-4.5	-3.2	0.0
jdepend	50.6	41.6	35.6	77.3	76.8	80.0	-1.2	-5.3	6.9
fop	43.9	37.7	30.5	65.9	68.6	67.4	-5.4	0.0	8.4
hsqldb	58.4	13.9	45.1	82.2	82.6	83.3	4.5	-2.1	0.1
mtrt	86.4	83.8	82.3	73.6	72.4	72.1	-20.1	2.8	3.3
volano	37.6	33.9	21.5	87.0	82.4	85.2	15.0	-0.7	27.8
psjbb	59.0	42.3	34.3	77.1	64.7	64.0	6.2	-3.6	-0.9
average	58.5	46.8	44.4	76.6	74.7	75.3	-1.1	-1.4	5.9

Table 4.6: Statistics for all compactors obtained for yield prediction turned on, the GC skip threshold of 5%, the young-old ratio of 1%, and for minimum heap sizes. Columns 2–4 show percentage execution time reduction due to GC skipping in YP. Next, in Columns 5–7 we report the percentage of skipped (unproductive) GCs. Reduction in maximum GC pause times is shown in Columns 8–10. The last row reports average values across the benchmarks.

Benchmark	Space Overhead [%]					
	Vs. Old Generation			Vs. Heap		
	CP	HS	MC	CP	HS	MC
beautyj	12.7	11.6	6.1	7.7	7.4	3.9
javaguard	23.4	22.0	20.5	5.8	5.5	5.4
jdepend	11.1	12.7	11.2	5.4	6.2	5.6
fop	17.3	24.4	17.5	4.1	6.3	4.2
hsqldb	5.1	9.9	4.9	3.4	6.7	3.3
mtrt	20.9	20.5	19.9	6.3	6.1	6.0
volano	9.2	8.6	8.9	4.4	4.1	4.4
psjbb	6.1	6.2	3.6	4.1	4.3	2.6
average	13.2	14.5	11.6	5.1	5.8	4.4

Table 4.7: Space overhead for all compactors obtained for yield prediction turned on, the GC skip threshold of 5%, the young-old ratio of 1%, and for minimum heap sizes. Columns 2–7 present space overhead (as percentage) imposed by GC skipping in YP, relative to the old generation size (Columns 2–4) and heap size (Columns 5–7). The last row reports average values across the benchmarks.

imum overhead is 3% on average for these programs). We set the GC skip threshold to 5% and the young-old ratio to 1%.

We first evaluate the impact that YP has on overall execution time by comparing the benchmark performance when prediction (and GC skipping) is enabled and disabled. In Table 4.6 (Cols. 2–4), we show the application throughput improvement for minimum heap sizes for each compactor. On average, across the benchmarks, we observe significant improvements in execution time: e.g. reductions of 59% for CP, 47% for HS, and 44% for MC, on average.

Cols. 5–7 in Table 4.6 show the percentage of GCs eliminated (the skip rate) on average for each program across heap sizes. The skip rate varies between 64% and 87%, and has an average of 75% for HS and MC, and an average of 77% for CP; YP is able to avoid most GCs in these programs.

Since YP eliminates unproductive GCs, it thereby increases minimum mutator utilization [43] and program performance. By doing so, YP also reduces the number of pauses an application experiences and increases the intervals between pauses. In the Cols. 8–10 in Table 4.6, we report the impact that YP has on maximum pause times. YP tends to increase pause times since when multiple GC are skipped, the heap size becomes larger, and the collection that is finally performed imposes a longer pause (while being more productive). Occasionally, however, YP skips an expensive compaction with the net effect of reducing the maximum pause time. On average, in CP and HS, YP increases maximum pauses by 1%, and in MC, it reduces them by 6%.

The trade-off that YP makes to achieve these performance gains is in predictor overhead (below 4%) and in heap space. Cols. 2–7 in Table 4.7 show the space overhead that YP imposes for each compactor as a percentage of the old generation size and heap size. Each skipped collection creates a temporary space overhead in the heap that is reduced or eliminated by the next conventional GC. This overhead results from skipping potentially multiple consecutive GCs. Relative to the old generation size the overhead is below 15%. The overhead does not exceed 6% relative to the total heap size.

We next present application throughput without (Figure 4.2) and with (Figure 4.3) YP and GC skipping. Each figure shows per-benchmark plots, each with 3 performance curves that correspond to CP, HS, and MC, respectively. We report average execution time (data points) and standard deviation (error bars) computed from 5 runs for each heap size. From the differences between the graphs in these two figures, we observe that YP consistently outperforms conventional GC for all three compactors across heap sizes.

Note that YP outperforms a system employing heap overprovisioning to run GC less often. Giving more space to HS, MC, and CP (as much as YP space overhead) does not lead to better execution times than YP obtains.

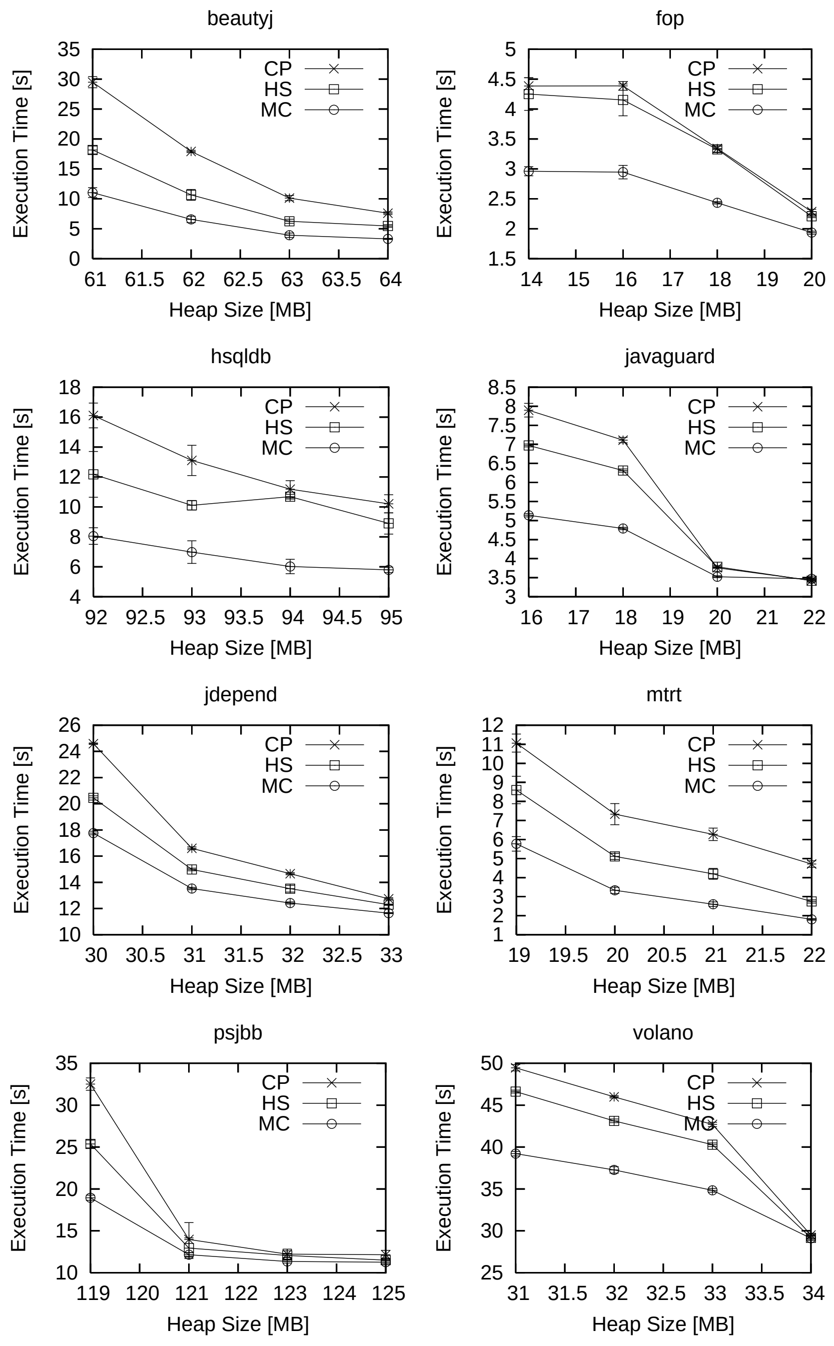


Figure 4.2: Benchmark execution times across heap sizes for all compactors. We report average and standard deviation (error bars) from 5 runs. Yield prediction is turned off.

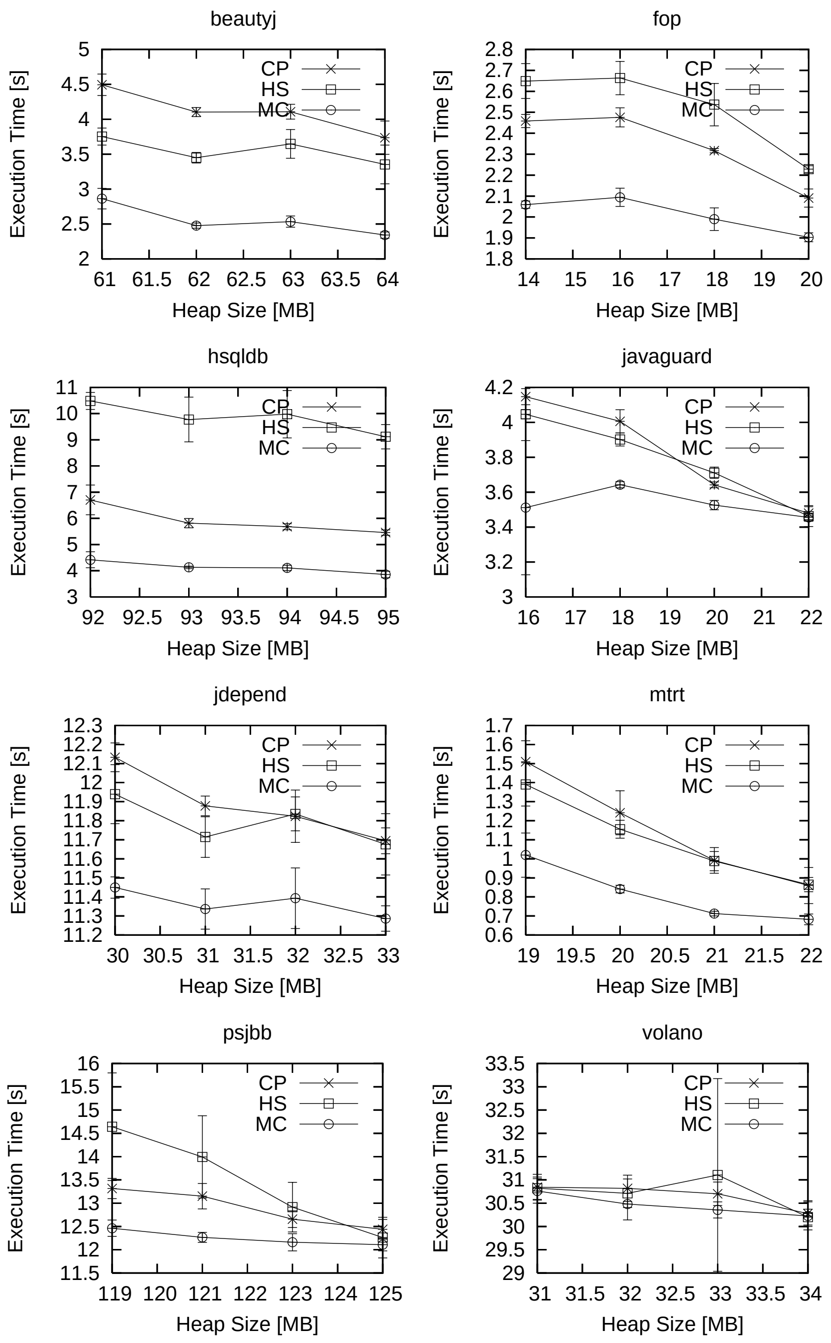


Figure 4.3: Benchmark execution times across heap sizes for all compactors. We report average and standard deviation (error bars) from 5 runs. Yield prediction is turned on, the GC skip threshold is 5% and the young-old ratio is 1%.

Thresh.	3%			5%			10%			Average		
GC	CP	HS	MC	CP	HS	MC	CP	HS	MC	CP	HS	MC
PEO	5.4	6.2	5.7	6.5	7.9	7.1	8.1	10.3	8.8	6.7	8.1	7.2
PEH	2.5	2.8	2.7	3.0	3.5	3.3	3.9	4.8	4.1	3.1	3.7	3.4
SOO	12.9	14.1	10.8	13.2	14.5	11.6	15.9	16.1	14.3	14.0	14.9	12.2
SOH	5.0	5.6	4.0	5.1	5.8	4.4	6.3	6.6	5.8	5.5	6.0	4.7
SR	70.6	71.8	70.1	76.6	74.7	75.3	83.2	80.5	80.6	76.8	75.7	75.3
ETR	55.7	41.8	42.0	58.5	46.8	44.4	62.3	53.5	47.2	58.8	47.4	44.5
MPR	-3.0	-0.7	4.2	-1.1	-1.4	5.9	0.7	0.3	10.6	-1.1	-0.6	6.9

Table 4.8: YP statistics for different GC skip thresholds (3%, 5%, and 10%) for each compactor (CP, HS, and MC). We report average values across benchmarks and heap sizes. Young-old ratio is 1%. All values are percentages. PEO is prediction error relative to the old generation size. PEH is prediction error relative to the heap size. SOO is space overhead relative to the old generation size. SOH is space overhead relative to the heap size. SR is GC skip rate. ETR is execution time reduction. MPR is maximum pause time reduction.

4.3.6 Other Parameter Values

We have also evaluated YP for the GC skip threshold of 3% and 10% to understand better how this parameter impacts application performance. Table 4.8 summarizes the results and compares them with the ones obtained for 5%. Overall, as the threshold increases, the prediction accuracy decreases, the space overhead increases, the skip rate increases, and we observe better performance gains. Thus, skip threshold selection is a space/time trade-off.

We have also investigated different values of the young-old ratio, a YP parameter which determines what proportion of the window between two subsequent GCs is considered young. The detailed YP evaluation we have presented thus far is for the 1% dead-young ratio. We have found this value to result in optimal prediction accuracy

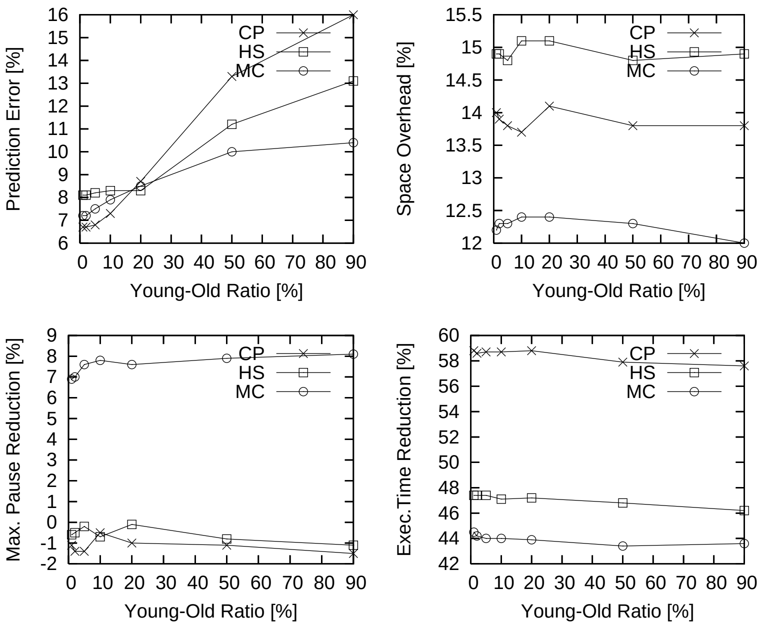


Figure 4.4: Impact of the young-old ratio on prediction error, space overhead, maximum pause time reduction, and execution time reduction.

(we have checked 1%, 2%, 5%, 10%, 20%, 50%, and 90%). Figure 4.4 shows the impact of the young-old ratio on prediction error, space overhead, maximum pause time reduction, and execution time. For each compactor, we report average values obtained across the three skip thresholds (3%, 5%, and 10%). Accuracy monotonically decreases when the young-old ratio increases (prediction error increases from 7% to 14%). This is because in a steady-state execution phase, programs allocate mostly short-lived objects. The remaining metrics are not overly sensitive to the young-old ratio. This is mostly because the prediction error never exceeds 16% for the ratios that we checked. Nonetheless, execution time reduction is worse for higher values of the young-old ratio.

4.4 Related Work

Static and dynamic prediction in the context of automatic memory management includes object lifetime prediction [21, 90, 48, 98, 134, 182, 139, 110] as well as heap size prediction [175, 77, 36, 174, 179]. In contrast, YP focuses on yield prediction. No prior work to our knowledge exploits page reference bits to predict GC yield accurately.

Like YP, MicroPhase [173] strives to improve the GC triggering mechanism to maximize the GC yield. MicroPhase recognizes phase boundaries and proactively invokes GC during phase transitions when many objects are expected to die. The system cooperates with the OS kernel to implement efficient profiling. In contrast, YP uses reference bits to predict GC yield and is therefore simpler while extracting the phase behavior implicitly.

Garbage collection hints (GCH) [39] is a profile-directed method for guiding garbage collection. GCH uses off-line profiling to identify favorable collection points in the program code where GC dynamically chooses between nursery and full-heap collections based on an analytical garbage collector cost-benefit model. In contrast, YP does not use off-line profiling and leverages hardware to make yield prediction.

The systems below are related to YP because they often actively interact with hardware and operating systems. However, they either do not leverage the mechanism of RR bits or do not implement yield prediction.

The Pauseless GC [46] is a parallel/concurrent compactor that uses specialized hardware and avoids pauses through read barriers, fast user-mode trap handlers, an additional intermediate TLB privilege level, and fast cooperative preemption via interrupts.

Numerous collectors leverage virtual memory operations. The Compressor employs both page mapping and protection. The Mapping Collector remaps free space in the address space. MarkCopy [136] reduces the memory footprint of a copying collector through on-the-fly (un)mapping of the copied pages.

Some collectors [77, 175, 80, 179, 174, 79] cooperate with the OS virtual memory manager to reduce the collector-induced paging. The Bookmarking Collector [80] records summary information about outgoing pointers from evicted pages to avoid accessing non-resident pages during compaction. CRAMM [175] and IV heap sizing [77] use VM paging behavior to predict and set dynamically the most-suitable, application-specific, heap size that adapts to changing memory pressure and avoid paging. The system described in [179] dynamically finds the optimal heap size by exploiting phase behavior to balance the GC frequency and collection cost as well as minimize the impact of page faults on performance. Many concurrent collectors also exploit virtual memory support [57, 101, 46, 122], which facilitates mutator conflict detection and exploitation of cache locality [101].

4.5 Summary and Conclusions

YP is a GC yield predictor that uses virtual page reference bits to accurately estimate the amount of reclaimable space in the heap. We incorporate YP into three state-of-the-art parallel compactors to verify its applicability to canonical heap layouts used by extant collectors. YP is simple and does not require changing the GC algorithm (only its triggering mechanism). YP enables better dynamic control over the space/time trade-off in MREs. We empirically evaluate YP using 3 compactors and 16 programs and find that YP consistently provides good accuracy while imposing low time overhead. In applications with many unproductive GCs, YP significantly improves performance (by 44–59% on average) by skipping most GCs and incurring modest space overhead.

The text of this chapter is in part a reprint of the material as it appears in [161].

Chapter 5

Concurrent Collection as a Service: Improving Intra-Runtime Memory Management Performance and Programming Model Using Shared Libraries

In this chapter, we describe an approach to improving intra-runtime memory management by using OS support for shared libraries. Specifically, we discuss the design and implementation of a lightweight GC library, portable across runtimes and languages, and providing parallel, concurrent, and on-the-fly collection. The library can be integrated into existing or newly-built runtimes using a fine-grain, low-overhead C interface. Decoupling GC from other runtime components simplifies the programming model for runtime developers and increases system modularity and component reuse. At the same time, the library allows to improve the GC performance in runtimes that do not implement modern memory management subsystems.

5.1 Introduction and Motivation

Managed runtime environments (MREs, virtual machines, VMs) for high-level, object-oriented (OO) programming languages are increasingly complex, which makes them challenging to architect, extend, and understand. One of the most complex components in MREs is automatic memory management (garbage collection, GC). State-of-the-art GC algorithms, i.e. parallel, concurrent, and on-the-fly GCs [167, 99], capable of taking advantage of multi-core processors, are notoriously difficult to implement, especially in conjunction with other MRE components (loaders, compilers, schedulers, etc).

As a result, it is not uncommon for MREs to implement simpler GCs, often at the expense of scalability, interactivity (pause times), and performance. For example, most extant MREs for dynamic languages use single-threaded stop-the-world GCs (e.g. Ruby) and reference-counting GCs (e.g. Python, PHP) while better GC algorithms have been known for decades. Even some MREs for static languages still rely on dated GCs, e.g. the Mono runtime for C# uses conservative stack scanning and stop-the-world serial GC (and until recently it has been based on the Boehm GC).

One way to address GC complexity is to decouple, modularize, and facilitate reuse of GC implementations [35, 34, 29, 28, 88]. We investigate the design and implementation of a portable GC library (which we call GC as-a-service (GaS)). GaS represents a new point in the GC design space because of its unique combination of goals:

- cross-MRE, cross-language GC library for static/dynamic languages,
- support for modern GC (concurrent, on-the-fly) for *cooperative* MREs (unlike Boehm GC [34]),
- GC-MRE decoupling (unlike recent on-the-fly GCs [61, 62]),
- low-overhead interface using C-based native API (unlike MMTk and GCTk [28, 29]).

We aim at increasing the GC quality and decreasing the GC engineering effort by code re-use, modularity, and separation of concerns. GaS decouples GC from other runtime components and exposes a fine-grain API for use by GC-cooperative runtimes of different programming languages for heap memory management. GaS provides concurrent, on-the-fly GC and avoids moving objects for use as a precise or conservative collector. We adapt the GC algorithm to avoid tight-coupling with the runtime in order to maximize portability and simplify GaS integration. GaS strives to minimize assumptions/restrictions regarding memory management in MREs.

We employ the GaS library within production-quality MREs for Java (HotSpot JVM) and Python (cPython) and compare GaS GC against state-of-the-art GCs. Our empirical evaluation includes concurrent, parallel, tracing GCs as well as hybrid tracing/reference counting GCs. We discuss the trade-offs we make with the GaS design and their performance implications. We also investigate the performance of other approaches that provide GC across languages such as those that cross language boundaries and that employ a single MRE for multiple languages. Our experimental results show that using GaS as an alternative to tightly integrated GC introduces modest overhead and that GaS reduces pause times significantly for Python and Java programs.

In the next sections, we describe the design and implementation of GaS (Section 5.2), present the results of GaS empirical evaluation (Section 5.3), discuss related work (Section 5.4), investigate how newly-built runtimes can benefit from GaS (Section 5.5), and conclude (Section 5.6).

5.2 Design and Implementation

Figure 5.1 presents the high-level architecture of GaS. GaS provides a shared C library that is accessible via the GaS interface and that can be used by MREs for different languages (e.g. Java, Python, Ruby) to integrate garbage collection (GC) into the runtime. Each MRE dedicates some number of threads to GaS GC (concurrent, on-the-fly GC) and maps a virtual memory region which GaS manages. MREs also have

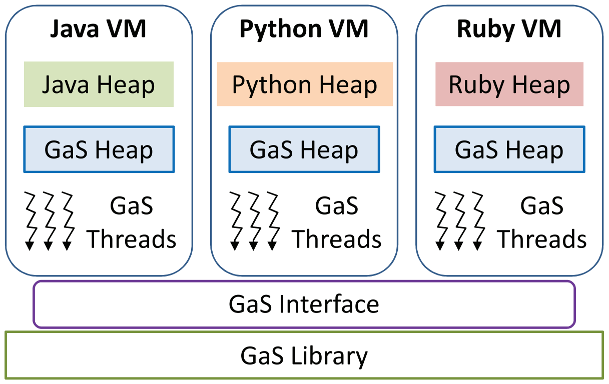


Figure 5.1: GaS architecture: multiple VMs share the GaS library. Each VM has its own heap and GC threads.

the option of allocating certain types of objects (e.g. immortal objects or internal data structures) in their private heaps and managing them independently of GaS.

We design GaS to support MREs for dynamic and static languages which implement diverse memory management strategies, including reference counting, tracing, object-moving, and non-moving GCs. Our goal is to enable GC portability at the library (i.e. binary) level (without recompiling the library, or modifying the GC algorithm).

The rationale behind GaS is to enhance modularity and separation of concerns in the design and implementation of MREs and to enable building new MREs from reusable components. GaS abstracts away the GC functionality, thus enabling construction of an MRE with a modern GC subsystem without expert knowledge about concurrent and on-the-fly GCs. By treating GC as a component, GaS facilitates research in other, non-GC, MRE subsystems. In addition, GaS enables integration of a high-quality GC into MREs that lack modern GCs, e.g. scripting language MREs that employ stop-the-world, single-threaded collectors (reference counting with cycle detection for Python and PHP, mark-sweep for Ruby). In Section 5.3 we show that even highly-optimized, sophisticated MREs, such as HotSpot JVM, can benefit from GaS.

GaS is a parallel (i.e. uses multiple GC threads), concurrent (i.e. collects most objects without stopping the application threads (mutators)), and on-the-fly (i.e. stops one thread at a time) GC. The rationale behind this configuration is that concurrent, on-the-fly GCs are difficult to implement, thus it is practical to provide such GCs as a service/library. In addition, many MREs are latency-sensitive, e.g. Ruby is used for server-side scripting and its stop-the-world GC is a limiting factor – concurrent GCs avoid stop-the-world collection which can introduce large pauses. Finally, as multi-core processors become ubiquitous, concurrent GC is increasingly suitable for fully utilizing and extracting high performance from modern systems.

GaS does not move objects because some MREs (e.g. Python) assume that object addresses remain constant and others (e.g. Mono) require support for object pinning and conservative root scan. GaS uses free-list allocation and thread-local allocation buffers (TLABs) for fast, unsynchronized, bump-pointer allocation in the common case. TLABs are vital for supporting multi-threaded MREs.

The GaS GC algorithm is an adaptation of extant snapshot-at-the-beginning (SATB) on-the-fly GC [61, 62, 60]. Our extensions decouple GC from the MRE and simplify the MRE-GC interface on the MRE side. Existing on-the-fly GCs rely on system-wide

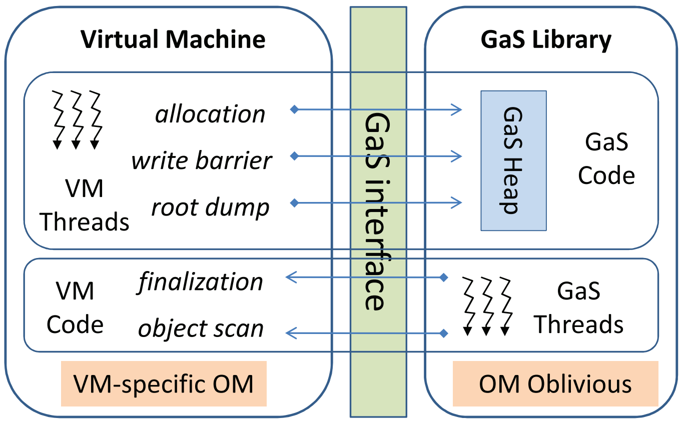


Figure 5.2: GaS interface. The upper part shows how a VM calls into the GaS library. The lower part lists the GaS callbacks.

handshakes with mutator threads and maintain per-thread buffers to implement write barriers and to determine quickly if another marking iteration is needed [61, 57]. GaS avoids such tight-coupling and moves GC logic out of the MRE as much as possible.

5.2.1 GaS Interface

Figure 5.2 depicts how MREs interact and cooperate with GaS. An MRE first initializes the GaS library by specifying the number of GC threads, TLAB size, and GC threshold (percentage heap usage that triggers a GC), and by providing a mapped virtual memory region for the GaS heap. The GaS interface consists of operations performed by MRE threads (allocation, write barrier, and root dump) and by the GaS threads (finalization and object scan).

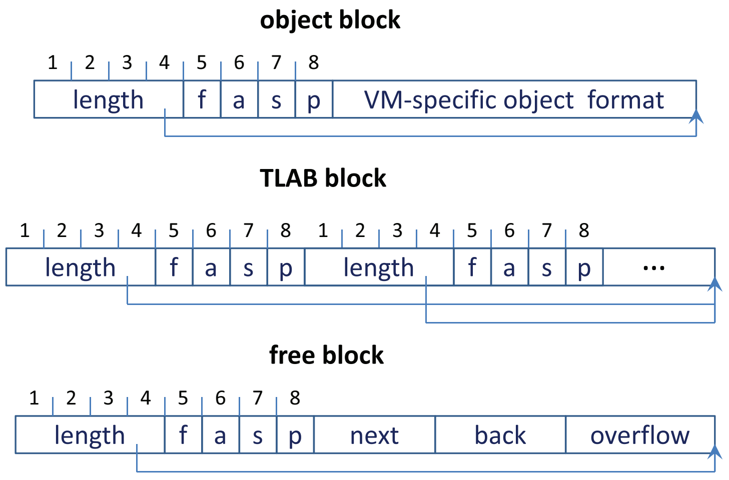


Figure 5.3: Block format in the GaS heap. There are three block types: object, TLAB, and free.

An MRE requests TLABs from GaS and performs most allocations within a TLAB. To allocate large objects, an MRE requests a TLAB of a specific size and then proceeds to intra-TLAB allocation. The GaS protocol for allocation and write barrier (described in detail in Section 5.2.2) is kept to a minimum so that the compiler can inline this code at allocation and reference store sites.

Before each GC, GaS requests a root dump. An MRE responds to this request by identifying objects (for GaS to mark) in the GaS heap that are reachable from thread stacks, global memory areas, and/or non-GaS generations. GaS invokes MRE-provided callbacks to scan objects for references and to indicate that a particular object is about to be reclaimed (to support finalization).

5.2.2 Heap Layout

We divide the GaS heap into blocks. Each block starts with a header, whose format is shown in Figure 5.3. The header size is one machine word (we assume 64-bit words) so that it can be atomically loaded/stored. GaS supports fully-concurrent unsynchronized sequential scans over heap blocks.

There are three block types: an object block, TLAB block, and free block. The block header consists of 5 fields: block length (4 bytes), block format (f, 1 byte), and three 1-byte GC flags: recently-allocated (a), scanned (s), and pending (p). We make each field at least 1-byte in size so that we can use atomic read/write (most architectures support single-byte atomic memory access but do not support bit-wise atomic access).

Object blocks are followed by an MRE-specific object representation, which is not interpreted by GaS. Thus, GaS adds one word of space overhead per object. GC flags have meaning only for object blocks. New objects have their recently-allocated flag set. Whenever the GaS GC marks a live object, it sets its scanned flag. Objects with their pending flag set will be scanned by the collector.

We initialize each word in a TLAB block so that we can treat it as the start of a new, shorter TLAB. For example if the first TLAB word contains length = 8, then the second TLAB word contains length = 7, etc. This approach enables atomic allocation of objects in TLABs. To allocate an object spanning 5 words, we simply store a new object header (with length = 5) at the beginning of the TLAB. Such a store happens atomically and the remaining part of a TLAB immediately has the right TLAB header (with the correct, shorter length). Thus, an ongoing concurrent heap block traversal cannot be confused by object allocation when we transition from a TLAB block to an object block. In addition, object allocation amounts to a single word store which the compiler inlines.

The length of object/TLAB blocks does not use the entire machine word. However, the limit of 16GB per object is typically sufficient in practice (e.g. in Java an object cannot exceed 16GB). Free blocks can use larger length values by storing their actual length in the overflow field (which has machine-word width).

When a TLAB fills up, we retire it (we insert a dead object into the remaining free space) and replace it with a new TLAB. TLAB allocation, like all freelist operations, employs synchronization. The freelist is a double-linked list of free blocks.

GaS uses a conditional SATB [61, 62] write barrier, that it executes before each store. The barrier first loads the previous pointer value (about to be overwritten by a store), checks if it belongs to the GaS heap, and if so sets the pending flag on the corresponding object. For example, before a store $*p = v$ happens we execute:

$if\ (is\_in\_gas\_heap(*p))\ then\ set\_pending\_flag(*p);$

For efficient heap membership checks, the MRE should map the GaS heap above or below all other object regions in an MRE – in such a setting a single border comparison suffices.

5.2.3 GC Algorithm

GaS GC comprises four concurrent phases: flag clearing, root dump, object marking, and object sweeping. GC threads use barrier synchronization to meet at subsequent GC phases. GaS imposes no pauses if an MRE is capable of performing a root dump without halting the mutator threads. A new GC cycle starts once the heap usage crosses the specified GC threshold.

We do not use a marking bitmap but instead mark object headers (the scanned flag) directly. This enables us to avoid atomic compare-and-swap (CAS) operations during marking because one byte can be stored atomically. Since we do not synchronize GC threads during marking, multiple GC threads may end up scanning the same object – we find that this happens rarely and we mitigate it via dynamic load balancing among the GaS GC threads.

Flag Clearing Flag clearing is a concurrent phase where a single GC thread traverses over the heap blocks and clears the GC flags. This step has a similar effect to activating the snapshot mode in extant SATB GCs [61, 62]. However, in GaS, the snapshot mode is active all the time, meaning that all objects are allocated live (the recently-allocated flag set) and mutators always use a SATB write barrier (setting the pending flag for objects whose incoming pointers are overwritten). This approach simplifies the MRE-

GC protocol and decouples GC and an MRE (no handshakes, state-dependent write barriers, etc. are required).

During flag clearing, GaS computes a balanced heap partitioning used in the subsequent, parallel heap traversals. GaS divides the heap into equal-size chunks at block boundaries. In later traversals, each GC thread uses its own chunk only.

Root Dump In the second GC phase, an MRE finds roots into the GaS heap and reports them to GaS by setting the pending flag for root objects. Depending on the MRE, root dump may require scanning registers, thread stacks, and global memory areas. An MRE may need to stop the mutator threads to find roots. Since GaS is an on-the-fly GC, an MRE is allowed to stop one thread at a time to avoid long pauses. In Sections 5.2.4 and 5.2.5, we describe how root dump can be done efficiently in MREs using tracing and reference counting, respectively.

Marking Object marking is parallel and concurrent. Due to concurrent object mutations, GaS occasionally performs several marking iterations before converging to a stable live object graph. In each iteration, every GC thread scans its own heap chunk for objects with the pending flag set. If no such objects are found by the concurrent block traversal, the marking phase is complete. Pending objects that GaS finds are recursively (using depth-first search) scanned and marked (by setting the scanned flag). Recursive marking stops on already-scanned objects (potentially marked in previous

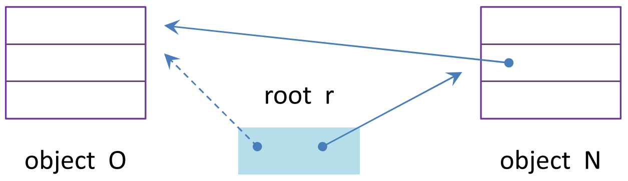


Figure 5.4: Root updates and concurrent marking. Since root updates are not captured by a write barrier, we use different marking stopping conditions during the first and later GC iterations.

marking iterations). GaS uses dynamic load balancing during marking (randomized work stealing) for scalability. GaS marks objects in-place (i.e. uses object headers) and, unlike some SATB GCs, does not use per-mutator marking buffers (to further decouple GC from the threading subsystem).

During the 2nd and later marking iterations, recursive marking stops on already-marked objects and on recently-allocated objects (the 1st iteration stops only on already-marked). This guarantees GC termination. Assuming there is N objects in the heap when the GC cycle starts, and all new objects are flagged as recently-allocated, GC will finish after N iterations at most. In practice 2 or 3 iterations suffice.

Figure 5.4 explains why this strategy is correct, i.e. it cannot lead to leaving some live objects unmarked. Since we stop the 2nd and later iterations of marking on recently-allocated objects, we need to guarantee that it is impossible that a recently-allocated object has a pointer to a live object that is otherwise unreachable and is not flagged as pending. Note that this is possible during the first marking, when we mark from roots. Consider an example in Figure 5.4. Root r initially points to object O.

Then, object N is allocated, and a pointer in N is set to point to object O. Next, root r is updated to point to N. Now we have a configuration where O is reachable only through N. Note that N is recently-allocated and still needs to be scanned. The reason for this is that our snapshot write barrier (SATB WB) does not capture root pointer updates (only heap pointer updates). However, the 2nd and later marking iterations ignore roots and mark from pending objects only. Thus, the newly-allocated objects do not have to be scanned once the first marking iteration completes. Reconsider our example in Figure 5.4 but assuming that r is not a root but a field in a heap object. On r update, object O is flagged pending and thus will be scanned by GC even if we do not scan N.

Sweeping Sweeping is parallel and concurrent. Each GC thread scans its heap chunk in an attempt to find a potentially-free block (i.e. either a freelist block or a dead object). This step is done without synchronization with mutators which perform concurrent allocation and might use free blocks in the meantime. Once a GC thread finds a potentially-free block, it acquires the freelist lock and continues scanning as long as it encounters reclaimable blocks (dead objects or free blocks). If the GC thread finds a contiguous region of sufficient length, it coalesces the region into a single free block and adds it to the freelist. Immediately prior to that, the thread invokes the finalizer on all dead objects. If a finalizer resurrects an object (the MRE $finalize$ callback indicates this to GaS), then the object will be finalized again once it becomes unreachable next time. Finally, the GC thread releases the freelist lock and looks for another potentially-free block in its chunk.

5.2.4 Tracing GC

Incorporating GaS into tracing MREs is relatively straightforward because such MREs already implement support for object scanning, root dump, and asynchronous finalization. Generational MREs in addition support card tables/remembered sets and write barriers.

In generational MREs, we extend the card table (or remembered sets) so that it is possible to quickly find not only inter-generational pointers but also pointers into the GaS heap. A minor collection then suffices to implement the root dump operation in GaS.

In non-generational MREs, we add a write barrier that captures pointers leading into the GaS heap as they are created. For each reference store we check if the new pointer points into the GaS heap and, if so, flag the object it points to as pending. After the flag clearing phase, GaS concurrently scans the memory regions in the MRE that might contain GaS roots, and relies on the write barrier to deal with roots that go by GaS unnoticed during the scan.

5.2.5 Reference Counting GC

Reference counting MREs associate a reference count with each object and rely on two operations: $incref$ (increment the count) and $decref$ (decrement the count) to detect and reclaim dead objects. Such MREs cannot reclaim cycles unless cycle detection is run periodically. Each reference update in the heap or on the stack invokes $decref$ for the old reference value and $incref$ for the new reference value.

To integrate GaS into a reference counting MRE, we make the $incref$ and $decref$ operations conditional. For pointers belonging to the GaS heap that point to an object in the GaS heap, we do not use reference counts. In all other cases $incref$ and $decref$ have their original semantics. In particular, outgoing and incoming pointers in the GaS heap are subject to reference counting and so are pointers outside of the GaS heap.

In this design, all objects in the GaS heap whose reference count is non-zero are roots for GaS GC (because they are pointed to from outside of the GaS heap). Thus, the root dump operation amounts to a concurrent scan of the GaS heap in search of objects with non-zero reference counts. Note that no pauses are required for a root dump. To deal with the race condition that might hide a root from GaS, we introduce a write barrier in $incref$: if the reference count goes from 0 to 1, we flag the object as pending. Thus, if a root scan sees reference count of 0, which later becomes 1, we do not miss a root.

The SATB write barrier piggybacks on $decref$ and is only needed in case the decrement is performed in the GaS heap. In addition, we modify $decref$ so that it does not call the object finalizer if the reference count drops to 0 in the GaS heap (GaS calls finalizers during sweeping).

5.2.6 GaS Extensions

Although GaS is a non-moving GC, we can extend it to perform (non-moving) generational collection. Instead of physical partitioning of the heap, we employ logical partitioning. Each object has an age field, incremented during each GC cycle until the object becomes old. Minor GCs mark only young objects and stop on old objects. A write barrier identifies old objects that contain pointers to young objects. Thus, the overhead of marking is significantly reduced. The sweeping cost, however, is still proportional to the heap size, as young and old objects are not physically separated.

To support conservative GCs, we extend GaS with an object-start array that enables GaS to quickly determine if a given address is the start of an object. GaS does not need to update pointers thus conservative roots do not pose a problem. GaS computes the object start array during the clearing phase and uses it during the root dump phase.

5.2.7 Implementation Details

We have implemented the GaS library in C and have integrated it into the HotSpot JVM 1.6 and cPython 3.1. The HotSpot JVM uses a generational heap layout while cPython employs hybrid reference counting/cycle detection. Both VMs use 2-word object headers. HotSpot employs safepoints for root scan, which halt all mutators, and uses a three-level, circular, unified object/class model.

Our implementation of the GaS GC assumes sequential consistency, i.e. there is some global order on writes and all threads see the same order. We use memory fences after the root dump phase to ensure store visibility. We use POSIX synchronization primitives (barriers, mutexes, and condition variables).

In HotSpot, we inline the GaS write barrier and object allocation in the template interpreter and in the code generated by the server (C2) compiler. We map the GaS heap at the constant border above all other generations, which reduces the membership checks to comparing a register with a constant. We use minor GC (based on parallel copying in the young generation) to find roots in thread stacks. For roots in other generations, we perform concurrent generation scan and introduce a write barrier to capture pointers into the GaS heap. We have found this approach to result in shorter pause times than if we instead leverage card tables (we discuss these alternatives in Section 5.2.4). We use the GaS heap for the young and old generation and leave the permanent generation as part of the MRE-private heap.

In cPython, we extend C macros INCREF and DECREF to implement conditional reference counting. We synchronize the GC and the VM interpreter after root dump and before marking by acquiring and immediately releasing the global interpreter lock (to ensure all write barriers have finished executing). cPython does not have safepoints and thus GaS imposes no pauses. Note that regular cPython does impose pauses for (1) cycle detection and (2) whenever freeing large data structures after $decref$. The GaS heap is located at a fixed precompiled address in the virtual memory. We implement GaS support in cPython for a single data structure: the binary search tree, which is sufficient to evaluate GaS using our benchmark described in detail in Section 5.3.

5.3 Experimental Evaluation

A primary goal of our experiments is to show that a cross-language, cross-runtime GC that is implemented as a C library, offers competitive performance (in terms of application execution time, GC pause times, and other GC metrics) compared to tightly-integrated VM-specific collectors in production-quality VMs. We find that GaS significantly reduces pause times and introduces modest overhead on overall execution time. In this section, we also investigate the tradeoffs associated (i) with the way GC is integrated into a runtime systems (built-in vs. a native/non-native library) and (ii)

GC	G1	CMS	RC/CD	GaS
concurrent	yes	yes		yes
on-the-fly				yes
parallel	yes	yes		yes
moving	yes	yes		
tracing	yes	yes	yes	yes
reference counting			yes	
generational	yes	yes	yes	

Table 5.1: High-level comparison of the GCs that we evaluate.

with different GC designs (generational vs. non-generational, moving vs. non-moving, concurrent vs. stop-the-world).

We first compare GaS to state-of-the-art GCs in the C-based runtimes for Python and Java. We use cPython (`http://docs.python.org/py3k/`) and the HotSpot JVM (`http://openjdk.java.net`). cPython implements a single-threaded Reference Counting [45] with generational stop-the-world Cycle Detection (RC/CD) [13]. The HotSpot JVM implements two concurrent, parallel, and generational GCs: Garbage-First (G1) [57] and Concurrent Mark Sweep (CMS) [128]. Table 5.3 summarizes the main characteristics of these GCs compared to GaS.

RC/CD divides the heap into three generations. Once the number of objects with non-zero reference counts in the youngest generation reaches a specific threshold, RC/CD traces the object graph to find and free possible reference cycles within this generation. Survivors are promoted to the older generation. Generation $i + 1$ gets collected after the specified number of collections of generation i. RC/CD does not move objects and segregates object into generations logically (i.e. it maintains a list of objects in each generation).

CMS [128] is a mostly-concurrent incremental GC based on the mostly-parallel collection algorithm described by Boehm et al [34]. HotSpot JVM implements CMS in the old generation and overloads generational write-barriers to identify objects that are modified during concurrent marking (these objects must be rescanned to ensure that the concurrent marking phase marks all live objects). CMS imposes two pauses per GC cycle: for initial marking and for remarking. CMS does not move/compact objects except for promotion to the old generation and copying within the young generation.

G1 [57] is a concurrent GC designed to meet a soft real-time goal with high probability, while achieving high throughput. G1 performs marking concurrently but halts mutators during object evacuation. Marking identifies regions that contain few live objects and that can be evacuated within a given pause time limit (with high probability). Each region has an associated remembered set, which indicates all locations that might contain pointers to (live) objects within the region. At carefully scheduled points, G1 stops the mutator threads and performs an evacuation pause. G1 is generational – regions holding current TLABs are treated as young and always belong to the evacuation set. G1 opportunistically moves objects to gradually defragment the heap.

5.3.1 Methodology

For our experiments, we use a dedicated machine with a quad-core Intel Xeon and 8GB main memory. Each core is clocked at 2.66GHz and has 6MB cache. Our platform runs 64-bit Ubuntu Linux 8.04 (Hardy) with the 2.6.24 SMP kernel.

We use HotSpot JVM from OpenJDK 6 build 19 (released April 2010) compiled with GCC 4.2.4 in the 64-bit mode. Our configuration employs the server (C2) compiler, biased locking, and two concurrent GCs: G1 (garbage-first) and CMS (concurrent mark-sweep) in a generational heap. In case of CMS, the young generation uses a parallel copying GC [85].

For the Java experiments, we employ the DaCapo'08 [54] and SPECjbb'00 benchmarks. We use the default input for DaCapo and 1 warehouse with 75s runs for SPECjbb. We disable explicit GC invocation. For the Python experiments, we use the open-source cPython 3.1.1 (released in August 2009) compiled with GCC 4.2.4 in the 64-bit mode. Our Python benchmarks include PyBench (a collection of tests that provides a standardized way to measure the performance of Python implementations), a set of Shootout cPython benchmarks (`http://shootout.alioth.debian.org/`), and PyStone (a standard synthetic Python benchmark). Since there are no standard memory-intensive benchmarks for Python, we implement our own GC benchmark, called BST, which we model after SPECjbb. BST executes a number of iterations against a balanced binary search tree. Each iteration comprises 3 lookups, 1 insert, and 1 delete. This emulates realistic workloads by simulating an in-memory database.

We investigate the sensitivity of GaS to different parameter values across benchmarks. For the Java GCs, we vary four GC parameters: TLAB size, young generation size, number of GC threads, and GC-start threshold. We use the recommended values of this parameters (as described in the HotSpot documentation) for our detailed per-benchmark evaluation. For the Python RC/CD GC, we vary one parameter: the GC-start threshold which controls the frequency of cycle detection in the young generation. RC/CD has no other parameters that significantly affect GC.

We evaluate the Java and Python GCs using four main metrics: throughput (execution time), GC pause times (average and maximum), minimum mutator utilization (MMU), and minimum required heap size. We do so across a range of heap sizes starting at the minimum heap size to at least its double. Note that concurrent GC requires more heap space than stop-the-world GC due to delayed garbage reclamation and allocations happening during collection. In cPython RC/CD, there is no reliable standard way of setting the heap size, therefore we do not vary the heap size in this case. We repeat each measurement a minimum of 5 times and report standard deviation as appropriate (error bars in plots).

Bench-mark	Minimum Heap GaS [MB]	Minimum Heap vs.G1 [x incr.]	Minimum Heap vs.CMS [x incr.]	Execution Time GaS [s]	Execution Time vs.G1 [% incr.]	Execution Time vs.CMS [% incr.]
antlr	40	2.0	4.0	10.9	6.0	9.9
bloat	140	4.7	14.0	25.5	11.5	17.9
chart	100	1.0	3.3	25.7	8.9	9.2
eclipse	400	8.0	5.7	71.9	11.4	15.5
hsqldb	290	1.9	1.9	12.7	-1.6	-1.5
jython	80	4.0	2.7	29.9	4.8	7.8
luindex	120	2.4	12.0	25.6	4.8	4.4
pmd	250	3.6	4.2	20.3	12.1	16.7
xalan	80	1.3	0.4	23.6	29.0	24.6
average	167	3.2	5.4	27.3	9.7	11.6
				Throughput [kbops]	[% decr.]	[% decr.]
jbb	110	1.8	2.2	3.9	5.7	6.3

Table 5.2: Comparison of Java GCs: G1, CMS, and GaS. Columns 2–4 show the minimum required heap size and Columns 5–7 show execution time/throughput.

5.3.2 Java Benchmarks

Table 5.2 and Table 5.3 detail per-benchmark, GC metrics for GaS, G1, and CMS. These experiments use our baseline GC parameters. The TLAB size is 4kB, we use 2 GC threads, the GC-start threshold is 50% (i.e. collection starts once half of the heap is filled), and the young generation size is fixed at 8MB (the HotSpot documentation recommends the young generation size to be set to 4MB times the number of GC threads).

We next evaluate the impact of each GC parameter on the different GC metrics. When measuring pause times and execution time/throughput we use the minimum heap size that each benchmark requires to run under GaS, G1, and CMS.

Bench-mark	Average Pause			Maximum Pause		
	GaS [ms]	vs.G1 [x decr.]	vs.CMS [x decr.]	GaS [ms]	vs.G1 [x decr.]	vs.CMS [x decr.]
antlr	0.8	2.5	1.5	2.5	2.1	1.5
bloat	0.5	3.8	1.4	2.0	2.4	2.6
chart	0.3	12.0	6.2	2.0	4.8	3.0
eclipse	0.6	7.3	3.4	3.4	4.4	4.6
hsqldb	0.5	36.3	20.6	1.4	17.2	22.9
jython	0.8	2.3	0.8	3.9	1.2	1.5
luindex	0.3	7.5	3.7	1.5	4.2	3.1
pmd	0.3	27.2	15.0	1.2	31.9	18.2
xalan	0.8	4.7	4.2	4.1	1.8	2.8
average	0.5	11.5	6.3	2.4	7.8	6.7
jbb	0.5	12.9	4.3	1.9	7.2	6.3

Table 5.3: Comparison of Java GCs: G1, CMS, and GaS. In Columns 2–4 and 5–7 we report average and maximum pause times: for GaS in ms and number of times decrease relative to G1 and CMS.

Pause Times and MMU In Table 5.3 we report both average and maximum pauses (in milliseconds for GaS, and as number of times decrease relative to G1 and CMS). Across benchmarks, average pause times in GaS are shorter by 12x compared to G1 and 6x compared to CMS. Maximum pause times in GaS are shorter by 8x compared to G1 and by 7x compared to CMS (across benchmarks).

Figure 5.5 and Figure 5.6 show the minimum mutator utilization (MMU) plots for the benchmarks and GCs. MMU curves [43] lend insight into the distribution of GC pauses across program execution (we define this GC metric in Section 2.1).

We do not include GC write barriers when computing MMU – we only take GC pauses into account. GaS achieves better utilization than G1 and CMS for all benchmarks.

Throughput In the last three columns in Table 5.2 we show per-benchmark execution time/throughput for GaS and percentage overhead of GaS relative to G1 and CMS. Across the DaCapo benchmarks GaS imposes 9.7% overhead compared to G1 and 11.6% overhead compared to CMS. For JBB, throughput reduction due to GaS is 5.7% relative to G1 and 6.3% relative to CMS. GaS overhead is mostly caused by GC write barriers and is overestimated here because our implementation of the write barriers is not as optimized as it could be.

Figure 5.7 and Figure 5.8 show per-benchmark execution time as a function of heap size. Each plot starts at the minimum heap size. CMS and G1 have similar performance for our benchmarks. We do not observe significant execution time increase for minimum heap sizes typical of stop-the-world GC. This is because GCs run on separate cores and only slow the program down for short pauses during which little processing takes place.

Heap Size In Columns 2–4 we report minimum required heap size for each benchmark (for GaS in MB and for G1 and CMS as number of times decrease relative to GaS). GaS requires larger minimum heap sizes than G1 (by 3x on average) and CMS

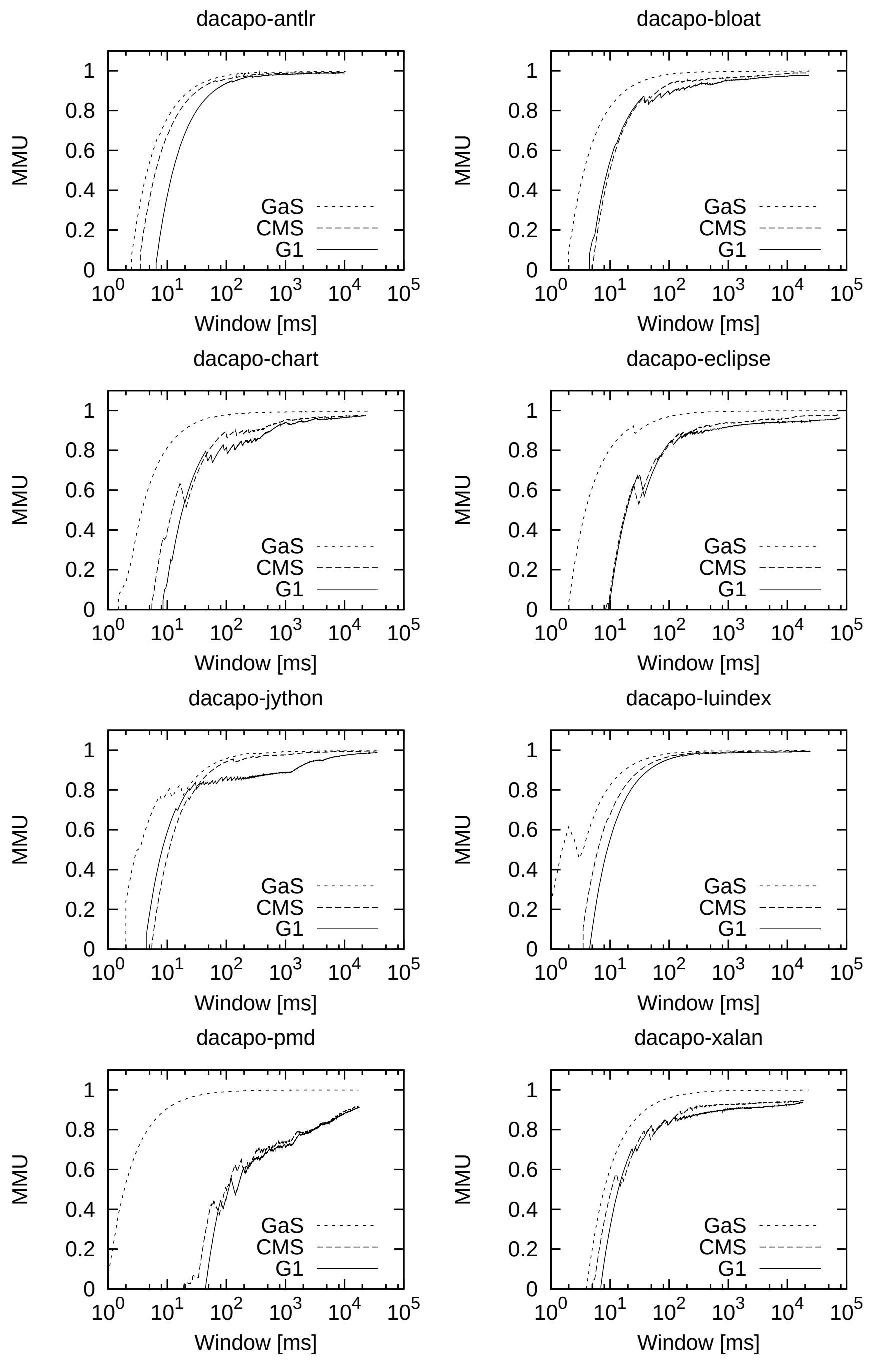


Figure 5.5: Minimum mutator utilization (MMU) for the client-side DaCapo benchmarks. We compare GaS with two HotSpot GCs: G1 and CMS. In all the plots, the x-axis (logarithmic scale) is a MMU window size (in ms).

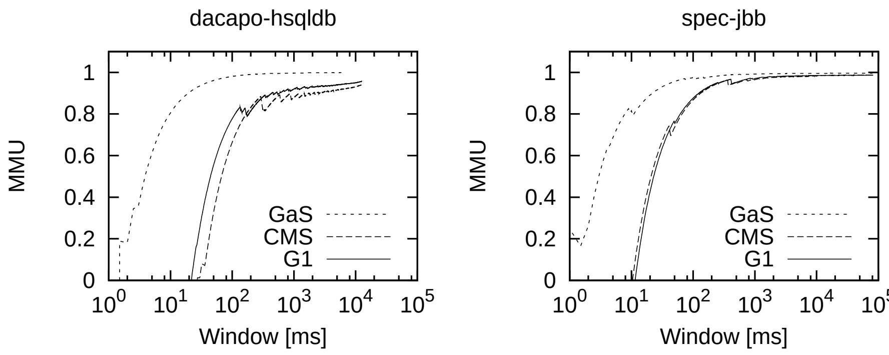


Figure 5.6: Minimum mutator utilization (MMU) for the server-side benchmarks: DaCapo hsqldb and JBB. We compare GaS with two HotSpot GCs: G1 and CMS. In all the plots, the x-axis (logarithmic scale) is a MMU window size (in ms).

(by 5x on average) because of three reasons. First, G1 and CMS are generational and thus tolerate allocation bursts better and place less pressure on the concurrent GC which executes for the old generation only. Second, GaS does not move objects and thus suffers from fragmentation (CMS uses a copying GC in the young generation and G1 performs opportunistic block-based compaction). Third, GaS adds a per-object header word, which may matter in benchmarks that allocate small objects. Each of these reasons is a consequence of a primary GaS design goal to be portable across runtimes and languages with different memory management subsystems.

Note that in case of concurrent GC, heap overprovisioning does not impact performance significantly (unlike in case of stop-the-world GC [161, 159]). That is, across all the benchmarks, giving G1 and CMS much more heap does not improve their performance.

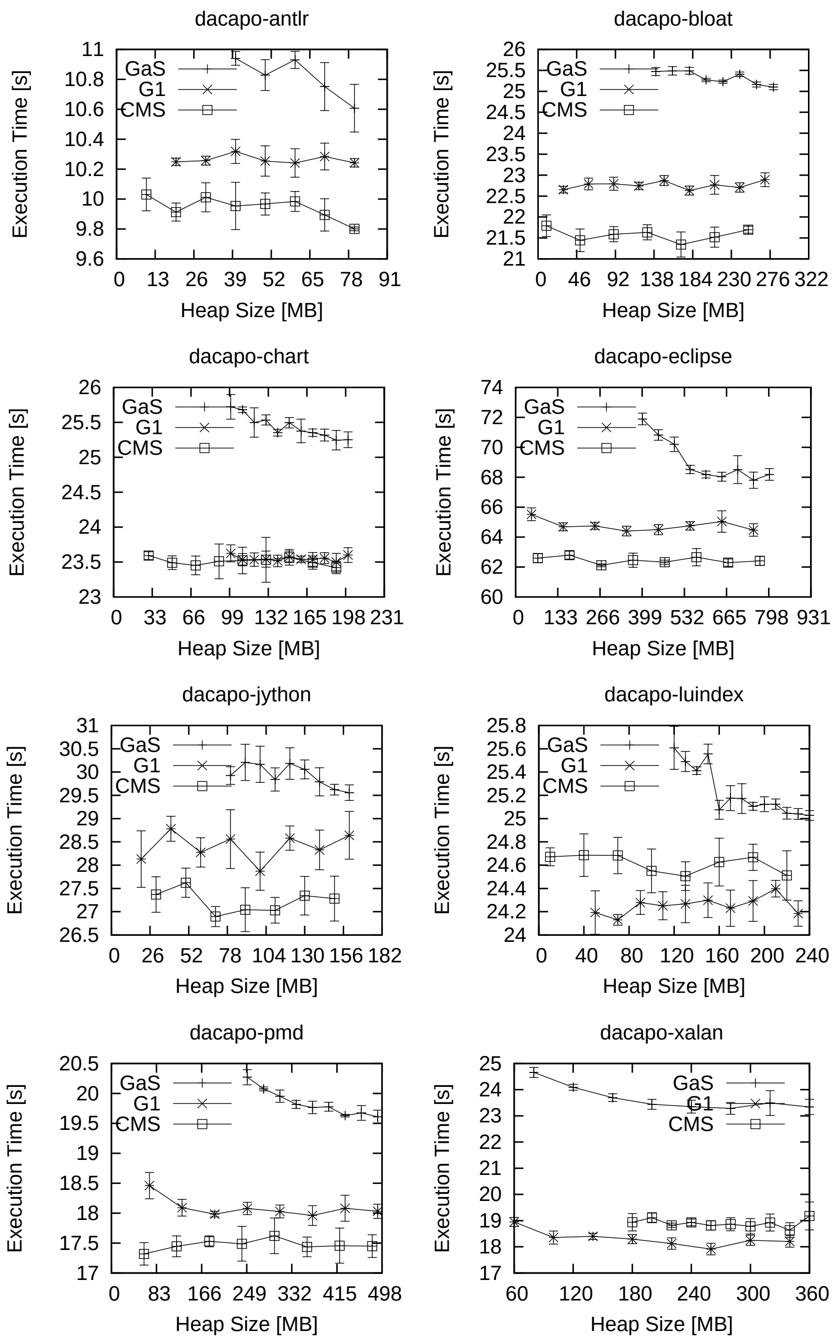


Figure 5.7: Execution time for the client-side DaCapo benchmarks as a function of heap size. We compare GaS with two HotSpot GCs: G1 and CMS. Each plot starts at the minimum heap size.

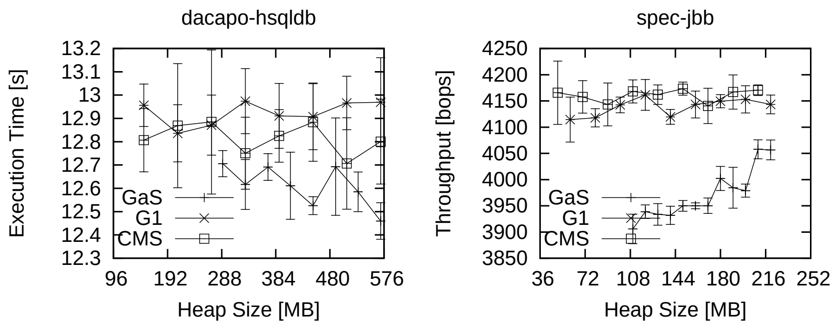


Figure 5.8: Execution time (for DaCapo hsqldb) and throughput (for JBB'00) as a function of heap size. We compare GaS with two HotSpot GCs: G1 and CMS. Each plot starts at the minimum heap size.

Sensitivity to GC Parameters To evaluate the parameter sensitivity of GaS, we vary the TLAB size between 1kB and 16kB, the young generation size between 2MB and 32MB, the number of GC threads between 1 and 3 (note that we have only 4 cores and we need to leave one core for the actual program), and the GC-start threshold between 20% and 80%. Our baseline values of GC parameters (reported previously) are medians of these ranges.

In Table 5.4, we present how our GC metrics (throughput, average and maximum pause times, and minimum heap) depend on GC parameters. The table consists of two parts. The first part (Rows 3–6) reports the GC metrics for GaS relative to G1 and the second part (Rows 8–11) relative to CMS. We vary one GC parameter at a time and keep the remaining 3 parameters at their baseline values. Column 2 corresponds to the baseline values of all 4 parameters. Each of the following Columns (3–10) reports the impact of one parameter: TLAB size, GC-start threshold, number of GC threads,

Met-ric	Base-line	TLAB [kB]		GC Threshold		GC Threads		Young Gen.[MB]	
		1	16	20	80	1	3	2	32
Relative to HotSpot G1									
TD	8.40	7.54	7.86	10.29	6.95	5.24	8.15	3.50	12.25
APD	10.12	9.76	10.43	12.77	8.73	14.77	8.29	4.32	31.34
MPD	8.26	7.58	8.34	7.58	9.33	10.76	7.85	5.25	20.89
MHI	3.07	2.30	4.69	3.07	3.86	3.11	3.10	4.63	1.81
Relative to HotSpot CMS									
TD	9.97	9.60	9.59	11.19	9.50	7.80	9.85	2.99	13.50
APD	5.52	5.19	5.57	6.48	5.81	6.69	4.44	2.10	13.45
MPD	7.67	6.50	6.94	6.53	8.29	8.59	8.05	3.80	21.68
MHI	5.05	3.33	8.11	5.10	4.60	5.23	5.27	6.26	4.64

Table 5.4: GC parameters' impact on the GC metrics in Java. Column 2 contains results for the baseline parameters: 4kB TLAB, 2 GC threads, 50% threshold, and 8MB young gen. Each subsequent column shows the impact of one GC parameter while the other 3 are kept at the baseline. Legend: TD: throughput decrease [%], APD: average pause decrease [x], MPD: maximum pause decrease [x], MHI: minimum heap increase [x].

and young generation size. We report GC metrics for two extreme values of each GC parameter. For each benchmark, we use the minimum heap size in which all experiments for the benchmark run. We report average results across benchmarks (DaCapo and JBB).

The young generation size has the greatest impact on all GC metrics. For small sizes (2MB), GaS degrades throughput 3-4% relative to G1 and CMS. For large sizes (32MB) throughput degradation is 12-14%. GaS converges to G1/CMS performance as G1/CMS approach non-generational GC.

G1/CMS pause times increase significantly for larger young generation sizes (up to 22-31 times longer than for GaS). For small sizes, G1/CMS pause times are 2-5 times worse than for GaS. This is because G1 and CMS are generational hybrids of concurrent and stop-the-world GC and trade throughput for pause times. Note that GaS does not have this tradeoff. Finally, large young generation sizes increase the minimum heap size in G1/CMS because during minor GCs more objects get promoted and, as a result, there is more pressure on the concurrent GC.

Dedicating fewer threads to GC in all collectors prolongs pause times and decreases throughput. TLAB size impacts only minimum heap size – large TLABs require that GaS uses more heap than G1 and CMS. This is because allocation rate is higher with large TLABs. GC-start threshold has only a minor impact on the GC metrics.

5.3.3 Python Benchmarks

To evaluate cPython hybrid GC, our BST benchmark creates both cyclic (collected by tracing) and acyclic (collected by reference counting) garbage. To create cycles we use self-referencing objects. We investigate 3 configurations: all-cyclic, all-acyclic, and 50% cyclic. Our main evaluation uses the last one. We have evaluated the all-cyclic and all-acylic configurations relative to the 50% cyclic one using our GC metrics. We have found that the all-cyclic configuration has shorter pauses (by 21-22%), larger minimum heap size (by 15%), and 3% worse execution time. The all-acyclic configuration has shorter pauses (by 49-56%), smaller minimum heap size (by 41%), and better execution

Met-ric	Base-line	GC-Start Threshold [number of young unreclaimed objects]					
		70		700		7000	
		RC/CD	Rel.	RC/CD	Rel.	RC/CD	Rel.
MH	17	7	2.4 [XI]	10	1.7 [XI]	34	0.5 [XI]
AP	0	0.2	–	0.8	–	5.4	–
MP	0	56.6	–	100.3	–	302.9	–
ET	39.3	38.0	3.6 [PI]	36.3	8.4 [PI]	35.3	11.4 [PI]

Table 5.5: GC metrics for GaS and Python RC/CD for different values of the young generation threshold in RC/CD (70, 700, and 7000). We report minimum heap in MB, average and maximum pauses, and BST throughput (time per 1000 iterations). Column 2 shows the results for GaS in its baseline configuration. The following columns compare GaS and RC/CD for different thresholds. Legend: MH: minimum heap [MB], AP: average pause [ms], MP: maximum pause [ms], ET: execution time [ms/10^3iters], XI: number of times increase, PI: percent increase.

time (by 5%). When RC/CD relies only on tracing, it imposes more overhead, uses 2x more heap, and has up to 2x longer pauses than when it uses only reference counting.

We allocate 15-level trees in BST. The live data set size does not impact RC/CD in Python because the cost of tracing in this GC depends mostly on the number of objects that are reachable from potential cycles (it does not matter if they are live or dead). The cost of tracing in GaS is proportional to the size of live data.

In Table 5.5 we report the GC metrics for GaS and RC/CD. Column 2 shows the results for GaS that correspond to our baseline GC parameters (2 GC threads, 50% GC-start threshold, and 4kB TLABs). We report the minimum heap (we instrument cPython to measure it), pause times, and execution time per BST iteration.

Columns 3–8 compare GaS with RC/CD for 3 different values of the main RC/CD parameter (the GC-start threshold). For its default value (700) GaS requires 1.7x more

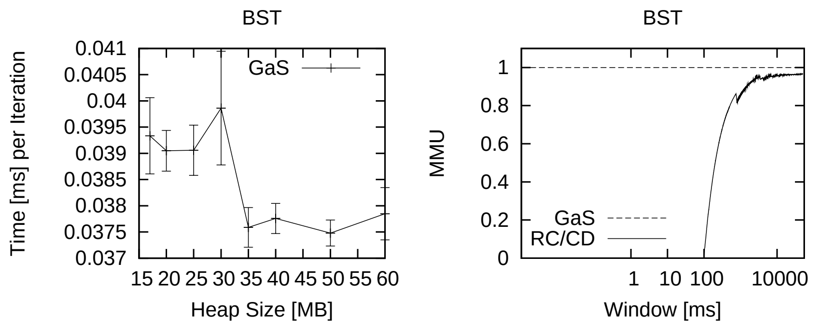


Figure 5.9: Results for Python and the BST benchmark. The left plot shows execution time per BST iteration as a function of heap size for GaS (in RC/CD heap size is not a GC parameter). The right plot is MMU for GaS and RC/CD. Note that since GaS does not impose pauses its MMU is at 1.0 across the window sizes.

heap and has 8% lower throughput relative to RC/CD. However, GaS imposes no pauses, while RC/CD does (up to 100ms, and 0.8ms on average).

Setting the GC-start threshold to 70 results in more frequent GCs in RC/CD. This results in shorter pauses (0.2ms on average and 57ms maximum), worse throughput (only 4% better than GaS), and lower minimum heap. Similar space/time tradeoffs can be observed when the young generation threshold is 7000. Now CD GC is relatively rare but each cycle is expensive. Pause times increase (5.4 ms on average and 303ms maximum), throughput improves (11% better than GaS), and the minimum heap increases (exceeding 2x GaS).

The left plot in Figure 5.9 shows how sensitive per-iteration execution time in BST is on heap size in GaS. BST throughput is 3% better for heap sizes that are 2 times the minimum. Heap size is not a GC parameter for RC/CD.

Benchmark	Execution Time [s]	% GaS Overhead
pybench	3.91	2.05
pystone	4.12	3.40
binary-trees	6.71	3.13
fannkuch	1.95	8.72
mandelbrot	15.48	2.13
meteor-contest	2.26	4.42
n-body	8.44	-0.59
spectral-norm	14.28	3.01
average	7.14	3.28

Table 5.6: Execution time overhead in GaS for standard Python benchmarks relative to RC/CD.

The right plot in Figure 5.9 shows MMU for GaS and RC/CD. Since GaS in cPython has no pauses, its MMU equals 1.0 for all window sizes. In RC/CD, the maximum pause time is 100ms (we use the default 700 GC-start threshold). RC/CD approaches GaS utilization for window sizes above 1 second. The MMU plots do not take write barriers/conditional RC into account (only pause times). In RC/CD we only measure pause times caused by tracing. Reference counting imposes negligible pauses in BST because whenever we delete nodes we free one node at a time.

Table 5.6 shows execution time statistics for the Python benchmarks. These benchmarks are not memory intensive and do not exercise GaS GC like the BST benchmark does. On average, the overhead of GaS extensions is 3%.

5.3.4 Overhead of Cross-Runtime Calls

We next investigate the performance overhead of other ways of integrating a GC into an MRE. We first consider the approach that implements the GC in Java (e.g. MMTk and GCTk) that is then integrated into a C-based runtime. We evaluate the cost of crossing the runtime boundaries. We measure the overhead incurred by the up/down calls through the Java Native Interface (JNI) – the mechanism through which Java and C programs interact. We consider the key GC operations: object allocation and object scan.

We implement object allocation as a Java method `Object allocate(int size)` which takes object size as input and returns the allocated object. We upcall this method from C via JNI. Object scan is represented as a native method `int scan(Object o, Object[] b)` whose arguments are a reference to an object to scan and a reference to a buffer for pointers found in the scanned object. The method returns the number of references found. We downcall this native method from Java using JNI. Since our goal is to measure the JNI overhead, the $allocate$ and $scan$ methods do not perform any processing: $allocate$ returns NULL and $scan$ returns 4 pointers. We duplicate both methods in C and call them directly from C (without gcc inlining) to compare direct calls with JNI calls.

We run 10 experiments, each consisting of 10^6 calls. On average, when compared to direct (but not inlined) C calls, JNI upcalls for $allocate$ are 76x slower and downcalls for $scan$ are 5x slower. Downcalls are faster than upcalls because the HotSpot JIT compiler optimizes native calls extensively. For 10^6 calls, upcalls for $allocate$ introduce 225ms of overhead and downcalls for $scan$ incur 92ms of overhead.

The DaCapo benchmarks allocate between 2.4 and 161 million objects (with the mean of 18 million) whereas the number of live objects during a GC cycle reaches between 2.8 thousand and 3.2 million (with the mean of 104 thousand). Thus, the JNI overhead for allocation can range from 0.54s to 36s of execution time. Similarly, the JNI overhead for scanning (assuming 25 collections per program execution) can range from 64ms to 7.4s of execution time.

Such overhead is likely unacceptable for C-based runtimes which typically are tuned for high performance. MRE-neutral, C-based GC library is both easier to integrate into such runtimes and offers significantly better performance.

5.3.5 Overhead of Runtime Layering

We next consider another alternative approach to using a single GC for multiple programming languages: runtime layering. In this study, we investigate the cost of using a production-quality Java runtime to host a non-Java language. In particular, we compare the performance of Python benchmarks for Jython 2.5.1 (a Python runtime that executes on top of a JVM – the HotSpot JVM in our case), versus using cPython v2.6.

We omit the raw data due to space constraints, and summarize our findings here. For pybench and pystone, Jython is 2.5x and 1.74x slower than cPython. For the shootout benchmarks (those which Jython supports without extensive benchmark modifications) Jython is 2.97x (meteor-contest), 1.34x (spectral-norm), 2.24x (fannkuch), 1.72x (binary-trees), and 2.22x (n-body) slower. On average, cPython is 2.1x faster than Jython.

Re-using a Java runtime (and Java GC) to implement runtimes for other languages introduces significant overhead (in addition to being complex and time-consuming from the engineering standpoint). An alternative, simpler, and more efficient approach to incorporating a modern GC and memory management subsystem into a new or extant C-based runtime is to use a GC library like GaS.

5.3.6 Lines of Code

We next compare GaS, HotSpot G1/CMS, and Python RC/CD using lines-of-code, to lend insight into the approximate implementation effort required for each GC. The GaS library is around 1100 lines of C/C++. The integration/glue code in both Python and HotSpot is around 200 lines.

The implementation of G1 and CMS in HotSpot is around 30,000 and 22,000 lines of C/C++. RC/CD in cPython is 8,400 lines of C (note that reference counting code is scattered across the whole runtime). This suggests that GaS GC library is simpler to implement than G1, CMS, and RC/CD. In addition, 200 lines of the GaS integration code is 2 orders of magnitude fewer than that which is required to implement a modern GC from scratch in an MRE.

5.3.7 Results Summary

We have compared GaS with two generational, concurrent GCs for Java and a hybrid tracing/reference-counting GC for Python. GaS significantly improves pause times and MMU across all benchmarks and GCs. GaS requires larger heap sizes and imposes modest execution time overhead because it is non-generational and non-moving (unlike G1 and CMS) and concurrent (unlike RC/CD). GaS is non-moving so that it is able to support runtimes (such as Python) that make assumptions about object addresses.

We also investigate the performance sensitivity to different GC parameters on the GC metrics. We find that GaS minimum heap sizes and throughput converge to G1/CMS and RC/CD once the GC parameters mitigate the generational advantage of these GCs. We measure the overheads associated with other approaches to implementing a GC in an MRE (via cross-language calls and via runtime layering) and find that using a GC library in C-based runtimes is significantly simpler and more efficient.

5.4 Related Work

The work most related to GaS is the Boehm GC [35, 34]. Boehm GC is a widely-used GC library providing a conservative collector for uncooperative runtimes (such as C and C++). Boehm GC supports stop-the-world serial and parallel collection. In contrast, GaS focuses on concurrent, on-the-fly GC for cooperative runtimes (precise roots, write barriers, TLAB allocation etc.) Moreover, the GC interface in Boehm GC essentially consists of two functions: GC_MALLOC and GC_REALLOC. GaS interface is more fine-grain to be able to leverage runtime type-safe mechanisms for object scanning, finalization, and root dump (GaS and the MRE cooperate to a greater degree).

GC frameworks such as UMass GC Toolkit [88], GCTk [29], and MMTk [28] are different from and complementary to GaS. The UMass GC Toolkit (designed in the context of persistent Smalltalk and Modula-3) focuses on generational copying stop-the-world GC algorithms. GaS addresses concurrent, on-the-fly GC. GCTk, and MMTk are GC frameworks written in Java, created in the context of the Jikes RVM. Their goal is to support a number of different GCs to enable their comparative evaluation and GC research.

GCTk/MMTk have been used for non-Java languages, although such porting is not well-documented in the literature. For languages other than Java, however, these frameworks require crossing the C-Java language boundary for each GC operation (or translation/reimplementation of the entire framework). Crossing the C-Java language boundary incurs high overhead (in Section 5.3.4 we investigate the overhead of such crossings) and is therefore impractical for C-based runtimes (of which most MREs are).

GaS takes an alternative approach – the GC library and interface are written in C and do not require execution of an additional managed runtime (such as a JVM) to implement and use GC. The MRE-GC interface in GaS also differs from GCTk/MMTk in terms of granularity and encapsulation. By taking an MRE-neutral approach, GaS can afford fine-grain MRE-GC library interaction. In contrast, GCTk/MMTk in non-Java-based MREs must either use coarse-grain MRE-GC library interaction or break library encapsulation (because of the high cost of cross-language calls). Since MRE-GC interaction in inherently fine-grain (allocation/scanning/write barriers are frequent), to achieve good performance, non-Java-based MREs must replicate the GCTk/MMTk GC implementation in the MRE. GaS supports efficient direct fine-grain calls between GC and a MRE while maintaining the library encapsulation.

GaS is also simpler and more lightweight than GCTk/MMTk (where the approach is to support as many different GCs as possible, including object-moving GCs). Unlike GCTk/MMTk, GaS focuses on concurrent, on-the-fly GC and takes into account all restrictions placed on GC by different MREs (e.g. non-moving GC in cPython). GaS uses a GC algorithm designed specifically for a portable loosely-coupled GC library. The approach in GCTk/MMTk is to design the interface so that it supports diverse extant GCs.

Another way of reusing a GC implementation between two MREs is to implement an MRE in a high-level language, e.g. Jython, JRuby are Python/Ruby interpreters that run on top of a JVM and use JVM GC. The two key issues with such MRE layering is performance overhead (we investigate this empirically in Section 5.3.5), and incomplete/incompatible standard libraries (due to the extensive engineering effort required to make layering work).

Another system, called CoLoRS [164], provides cross-language, type-safe object sharing using POSIX shared memory for MREs that execute on the same physical hardware at the same time and interoperate. CoLoRS uses concurrent, on-the-fly GC for the shared memory region that each MRE maps into its address space. The CoLoRS GC however is tightly integrated into its runtime, and defines a new object and synchronization model for shared objects that it manages. GaS adds per-object headers and relies on MRE-native object model and synchronization.

VMKit [71] is a framework that eases the development of high-level MREs and thus enables experimentation with new languages and MREs and/or new language features. VMKit consists of a low-level and a high-level layer. The low-level layer provides threading support, GC-based memory management, and a JIT compiler that translates language-independent intermediate representation of programs. The high-level layer defines such aspects as object model, type system, call semantics, and method dispatch. VMKit glues together LLVM for JIT support, MMTk for GC, and POSIX thread library for multi-threading. VMKit translates MMTk into the LLVM intermediate representation in its entirety. VMKit performance, however, is orders of magnitude worse than production systems. GaS is orthogonal to VMKit in that GaS can be used as a GC component in the VMKit framework. Note, however, that GaS can be integrated not only with MRE frameworks, but also with general- and special-purpose MREs for both dynamic and static languages.

XIR [154] is a compiler-MRE interface that separates the compiler backend from an MRE. An XIR extension mechanism allows an MRE to express the machine-level implementation of object operations. The interface has a modest impact on compilation time without reducing performance. GaS is similar to XIR in its overall goal however GaS targets GC and XIR targets JIT compilation.

The idea of modularizing an MRE motivates the design and implementation of LadyVM [70]. LadyVM links three third-party software components: LLVM, Boehm GC, and GNU Classpath, to implement a Java VM. Similarly to VMKit, LadyVM can use GaS as a replacement for its GC component to enable modern, high-quality, concurrent GC.

Compiler libraries like LLVM [102] and VPU [126] enable modular approach to integrating JITs into VMs. LLVM is a compiler infrastructure designed for compile-time, link-time, and run-time optimization of programs written in arbitrary programming languages. LLVM supports a language-independent instruction set and type system. VPU is a high-level code generation utility that performs most of the complex tasks related to code generation, including register allocation, and which produces good-quality C ABI-compliant native code.

JnJVM [152] is a modular JVM that supports dynamic addition or replacement of its own modules without service interruption and state loss. JnJVM uses dynamic aspect weaving techniques and component architecture. GaS could potentially be used in JnJVM as a GC module.

The Common Language Infrastructure (CLI) [112] is an open specification (ECMA335) that describes the executable code format and runtime environment for multiple, static, high-level languages to be used on different computer platforms. All CLI-compatible languages compile to the Common Intermediate Language (CIL), which abstracts away the platform hardware. CLI is similar to GaS in that it provides GC (among other services) for multiple languages but it differs in that CLI uses monolithic architecture with built-in GC. GaS provides GC in a form of a library and targets multi-language support via the provision of a cross-MRE GC.

GNU Classpath [73] is a GNU project to create free core class libraries for use with virtual machines and compilers for Java. The Classpath library can be used with different VMs – it has a similar goal to GaS but pertains to core classes (not GC) and targets Java (not multiple MREs).

XMem [160], Singularity [65], MVM [53], and KaffeOS [10] provide isolation and sharing between MREs or tasks/processes and implement a common memory management system across them. GaS GC differs from GCs in these systems in that it is modular, loosely-coupled, and portable across different MREs and languages.

5.5 Newly-Built Runtimes

The GaS library can be used not only to enhance GC in existing MREs but also when designing and implementing a new language and/or runtime. In order to investigate how GaS impacts the process of architecting a new MRE, we build a runtime for a new scripting language and use the GaS library to implement its memory management component. Our goal is to determine a minimal set of runtime capabilities and services that are necessary to support a pauseless, concurrent, on-the-fly GC in a multi-threaded environment. We design and implement MiniVM, a GC-cooperative MRE that is able to eliminate the negative impact of GC on program performance and interactivity, provided that there are enough spare processing cores available. Unlike extant GC systems (including real-time and hardware-assisted ones), the MiniVM GC neither imposes pauses nor significantly slows mutators down, while avoiding complex MRE architecture.

MiniVM interprets binary programs generated by a source code compiler. The MRE targets an object-oriented language with dynamic typing and native support for multi-threading based on share-nothing semantics and explicit communication via message passing over channels. The language supports user-defined classes, single inheritance, dynamic dispatch, static and instance methods in classes, global functions, extensibility via native C code, dynamic field addition at runtime, and closures.

The interpreter is a hybrid stack/register machine. Function code can access arbitrary stack locations within the current frame. Instructions have fixed size and take up to three operands that identify source and target stack/constant pool locations for a specific operation. We implement a switch-based non-threaded interpreter.

Threads share the global constant pool and the GaS heap. Per-thread data structures include: a growable stack, a bounded incoming message queue (FIFO), and a TLAB for heap allocation. For control we use the C stack so the runtime stack contains only pointers (no return addresses). This makes it easier to scan stacks concurrently.

In the calling convention that we use, the caller saves all arguments (left to right) on the stack and creates/destroys stack frames. The return value overwrites the last argument on the stack. We do not use a frame pointer – stack locations are addressed relative to the stack top. Stacks grow in fixed-size chunks as needed. Each stack frame, however, is always contiguous.

The builtin types include: string, array, integer, real, thread, and function. Each object can be evaluated to true/false and used in a conditional statement. We implement operators such as addition/subtraction as function calls. Operations on per-thread queues (enqueue and dequeue) block if the queue is full/empty.

User-defined types, strings, and arrays have variable size and grow on demand. This mitigates the impact of heap fragmentation caused by a non-moving GC. Although objects are always allocated as contiguous regions, they might expand later on, thus consuming otherwise-unusable small fragments. We use a single-word object header that contains a class pointer. The initial size of variable-size objects is determined based on constructor parameters (e.g. array length) or compiler-produced hints (e.g. that type T typically has f fields). Thus, despite the lack of static typing, the system is able to keep space consumption close to optimal for most objects. Each class has one optional superclass and a (potentially empty) set of instance/static methods.

Dynamic field access, method dispatch, and type lookup take place by name. We use hash tables with open addressing to implement objects, per-class method dictionaries, and loaded classes (MiniVM supports static class loading only). Method dispatch always takes constant time because we populate method dictionaries of each type with all the inherited methods (to avoid walking the class hierarchy on each virtual call).

Open addressing keeps data structures more compact than chaining and reduces the number of pointer dereferences.

We allocate permanent runtime data structures, such as classes, constants, and metadata in the C heap. The GC does not traverse them. The remaining objects are allocated in the shared GaS-managed heap. Memory management is the most complex subsystem in MiniVM. GC significantly impacts the design of all other components (even though GaS is a library that abstracts away the details of the GC implementation). To fully leverage GaS capabilities and implement pauseless GC, each runtime component must support asynchronous, concurrent, and precise root dump at an arbitrary point in time while the program threads execute. In addition, the runtime needs to use the SATB write barrier for pointer updates in the heap.

We design a GC protocol that enables scanning message queues and stacks of all active threads without introducing GC pauses. Each thread has a bounded message queue and a growable stack, which are freed on thread termination. To prevent concurrent queue/stack scanning while freeing them, we use a global lock that the GC acquires for root dump and the runtime acquires for each thread start and exit. Using a lock in this case is acceptable because the system does not guarantee how soon a thread begins executing instructions after its start and frees its resources after its termination.

Message queues are bounded and provide potentially blocking semantics. Therefore, the GC can scan a queue with a lock being held, again without introducing pauses that are unexpected by a program. The same lock is used to synchronize concurrent access to a queue. The cost of scanning a queue is fixed and short blocking does not change the programming model provided by message queues.

Stacks are growable and potentially contain multiple chunks. They must be scanned concurrently without any locking because their length is unbounded and blocking is not expected by programs for stack operations. We scan each stack sequentially (bottom-up) without any synchronization. To guarantee that we obtain a stack snapshot (on-the-fly GC requires a per-thread snapshot at some point after GC request and before marking) we use a write barrier that captures newly-written stack pointers. Thus, even if a sequential scan misses a root, it is still reported by the write barrier. Stacks never shrink in order to avoid freeing stack chunks while the GC scans them. We use a handshake with each program thread before scanning its stack (the GC waits until the thread finishes executing the current instruction). This is necessary because the first GC phase (flag clearing) might make recently-allocated objects appear unreachable. Once the current instruction finishes, pointers to such objects are already written to the stack and can be found during the stack scan.

MiniVM supports C extensions via native functions/methods. The system loads them via dynamic linking at run-time. We use OS-level POSIX threads. We maintain a pool of unique constant strings that identify fields, methods, and classes so that on hash table lookup we can determine string equality by address comparison and without parsing actual characters. We maintain a double-linked list of active threads.

The MiniVM instructions enable programs to manipulate the current stack frame, object fields, and array elements, access constant pool, allocate new objects, create new threads, send/receive messages, push/pop frames, call functions/methods, and take (un)conditional branches. Instruction operands (integer immediates) identify stack and constant pool locations by index.

Although the MiniVM code size is below 3000 lines of C, the system implements a large subset of modern runtime services such as multithreading, OO support, and concurrent, pauseless GC. The key advantage of building a new MRE that uses GaS, compared to incorporating GaS into an existing MRE, is the ability to design the runtime data structures in a way that can fully leverage modern on-the-fly GC and thus avoid GC pauses and high overhead, while simplifying the runtime (e.g. by avoiding safepoints).

5.5.1 GC Evaluation

Although the MiniVM GC imposes no pauses, it incurs certain execution time overhead due to write barriers and concurrent collection that introduces additional memory traffic as well as cache pollution. In order to evaluate the worst-case cost of GC in MiniVM, we implement a simple GC benchmark and use two MiniVM configurations:

one with no GC activity and no write barriers and one with constant GC activity and frequent write barriers.

Our benchmark allocates a large single-linked list, which stays alive, and is repeatedly reversed. If the list length is above the GC-start threshold then GC is active all the time. Otherwise, GC is never invoked. By reversing the list in place, we generate a large number of write barriers, both in the heap and on the stack. Thus, we are able to estimate the upper-bound for the GC and write barrier cost.

Our heap size is 100MB, the list has 800K elements, and is reversed 200 times. Each program run lasts at least 30 seconds. We use one GC thread (so that there are enough cores for the GC and program threads). We set the GC-start threshold to 1% and 99% and compile MiniVM with and without write barriers. We repeat each execution time measurement 10 times.

We find that the write barrier overhead is 4% and the concurrent GC overhead is 3%, which add up to 7% total overhead. Given that this is an upper-bound, in practice the GC overhead is likely to be around 3%.

5.6 Summary and Conclusions

GaS is a lightweight, cross-MRE, cross-language GC library that provides concurrent, on-the-fly, non-moving GC. GaS can be integrated into MREs for static (e.g.

Java) and dynamic (e.g. Python) languages via a fine-grain, low-overhead GC interface. GaS is a stand-alone C-based library for GC-cooperative MREs. GaS GC adapts the SATB algorithm for loose coupling between GC and an MRE. The GaS library makes no assumptions about object model, threading, JIT, and memory management strategy (tracing, reference counting, generations, etc.) in an MRE. We implement GaS and integrate it within production-quality MREs for Java and Python. Our experimental evaluation shows that in comparison to built-in, tightly-coupled GCs, GaS can improve pause times significantly and offers competitive performance even when compared to generational GCs. The GaS library reduces the development effort required for implementing a state-of-the-art on-the-fly GC. The library can be used as a modern GC component both in extant MREs and when building new MREs for new or existing languages.

The text of this chapter is in part a reprint of the material as it appears in [163].

Chapter 6

Type-Safe Sharing for Homogeneous Runtimes: Improving Cross-Runtime Memory Management Performance and Programming Model Using Shared Memory

In this chapter, we describe an approach to improving cross-runtime memory management by using OS support for shared memory. Specifically, we discuss the design and implementation of type-safe, transparent object sharing for co-located Java runtimes run as separate OS processes. We overview our extensions to such runtime services as synchronization, class loading, object allocation, and garbage collection, necessary to implement object sharing. Our experimental evaluation compares sharing with extant communication mechanisms available on the Java platform, such as RMI, JNDI, JDBC, serialization, and network sockets. Our results indicate that object sharing improves cross-runtime memory management in two ways. First, it enriches the programming model by introducing shared memory that enables communication without explicit message passing. Second, it increases throughput and decreases latency by up to several orders of magnitude compared to state-of-the-art J2SE/J2EE communication mechanisms by avoiding object serialization and network communication.

6.1 Introduction and Motivation

Developers today predominately build modern, enterprise, component-based, middleware for portable, distributed applications using type-safe, object-oriented languages, such as Java, which users execute within managed runtime environments (MREs). These MREs typically support garbage collection (GC), dynamic class loading, incremental compilation, as well as high-level threading and synchronization primitives, among other runtime services. One popular example from this application domain is JBoss, an application server that provides a complete implementation of the J2EE [93] specification, and that runs on top of the Java platform [95].

A common architectural design pattern employed by administrators of enterprise applications is multi-tier deployment that partitions the system into independent domains, typically run using separate MRE instances (OS processes). Such isolation improves reliability and helps to manage system complexity by fault containment and modularity. J2EE-based applications typically comprise at least three tiers: a web container (front-end presentation layer), an application server (business logic), and a database engine (back-end data source) [95, 24, 158].

Multi-tier decomposition, however, necessitates expensive inter-process communication (IPC) between MREs (isolated components). Since most general-purpose servers (e.g. web, application, database) are designed for online transaction processing (OLTP), in which many clients perform many short transactions simultaneously, communication overhead can constitute a significant portion of the observed, end-to-end response time (especially when multiple isolation units are involved).

To reduce the overhead of cross-MRE IPC, administrators commonly co-locate multiple tiers on a single machine. Co-location simplifies administration and configuration, enables efficient use of local network communication for IPC, and makes better use of multi-processor architectures through increased thread-level parallelism. Emerging multi- and many-core systems are likely to make MRE co-location increasingly commonplace.

Cross-MRE IPC mechanisms cannot depend on co-location, however, since MREs may alternatively be distributed across different cluster nodes or be migrated to achieve load balancing and more effective utilization of server resources, an increasingly important operation in virtualizing systems today [116, 120, 148]. Thus, MRE IPC employs high-overhead implementations of standard communication protocols, such as remote procedure calls and object serialization, regardless of the proximity of the communicating MREs. These protocols are not optimized for the co-located case because state-of-the-art MREs offer no support for cross-runtime sharing. At the same time, efficient inter-process communication mechanisms, such as shared memory, are widely available as a standard (POSIX) operating system service on most modern platforms.

To address the growing need for inter-runtime object sharing, we introduce support for transparent and type-safe, cross-MRE communication and coordination, called XMem. XMem is an IPC mechanism that enables object sharing between MREs co-located on the same machine and communication via extant distributed protocols when physically separated. XMem is transparent in that shared objects are the same as unshared objects (in terms of field access, synchronization, GC, and method invocation, among others), except that XMem disallows pointers from shared objects into MRE-private storage. To enable efficient object sharing, XMem manipulates virtual memory mapping to avoid indirection, i.e. all object references in the system are direct. Moreover, existing communication technologies, e.g. those employed by J2EE or network sockets, can use XMem without application modification.

XMem guarantees type-safety by ensuring that the MREs employ the same types for shared objects when the communication medium is shared memory. XMem is also compatible with core MRE services such as GC, dynamic class loading, and thread synchronization. XMem coordinates MREs through infrequent, synchronized global operations that include GC and class loading.

XMem provides direct object sharing via isolated channels between co-located MREs isolated as distinct OS processes that avoids the trade-offs inherent to previous approaches [53, 10] by enabling communication without serialization and data copying. XMem extends existing MRE services, abstractions, and libraries as well as introduces their cross-process equivalents, including parallel, cross-MRE class loading and garbage collection. At the same time, XMem maintains standard, portable interaction with the lower-level layers of the software/hardware stack.

We implement XMem in the open-source, production-quality HotSpot Java Virtual Machine. Our experimental evaluation, based on core communication technologies underlying J2EE, as well as using open-source server applications, indicates that XMem significantly improves throughput and response time by avoiding the overheads imposed by object serialization and network communication.

In the sections that follow, we describe the necessary support for object sharing, multi-threading and management of the shared memory segment (Section 6.2). In Section 6.3, we present our experimental methodology and empirical evaluation of XMem. Finally, we contrast related work (Section 6.4) and present our conclusions in Section 6.5.

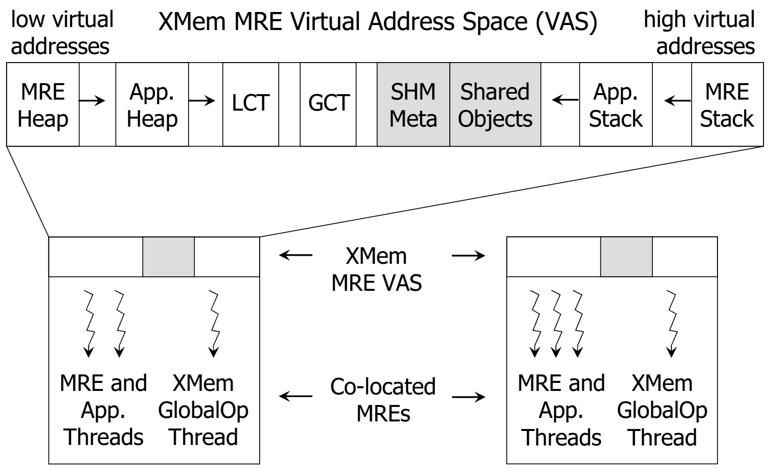


Figure 6.1: Co-located XMem MREs, and their virtual address spaces (VAS), that are attached to a shared memory segment (gray area). The shared region contains metadata (SHM-Meta) and shared objects and is mapped at the same virtual address in each MRE. The GlobalOp thread in each MRE performs infrequent global operations that XMem synchronizes across attached MREs.

6.2 Design and Implementation

The goal of XMem is to improve communication performance for enterprise-class, object-oriented, software systems, a popular application domain for web services. XMem enables transparent IPC via shared-memory between isolated MREs that are *co-located* on the same machine; such co-location of related processes is an increasingly common technique for the exploitation and better utilization of multicore systems. Using XMem, MREs share objects directly to avoid the overhead that is imposed by distributed communication protocols due to object marshalling and serialization.

To enable direct object sharing, XMem maps the shared memory segment at the same location in the virtual address space (VAS) of all attached MREs. Figure 6.1

depicts an example instance of an XMem system. Two MREs attach to the same shared memory segment (gray area of the VAS) to share objects. We refer to the VAS of each MRE that is not mapped to the shared memory segment (white area of the VAS) as MRE-private. XMem systems share per-instance, non-static data only – static (per-class) data is MRE-private since static fields typically record program-specific or MRE-specific state. Sharing of such fields can violate both type safety and inter-process resource isolation.

Since we map shared memory to the same virtual address in all MREs, objects within the shared memory have the same addresses in all MREs. To guarantee memory and type safety, we disallow pointers from shared objects to private objects via a write barrier (described further below), since the address space of the non-shared areas in MREs is independent and unrelated across MREs. Regardless of this constraint however, XMem MREs implement services, such as class loading, GC, allocation, synchronization, compilation, uniformly for shared and MRE-private objects, i.e. XMem provides object-level transparency.

Key to enabling such transparency efficiently is that (i) the internal representations of object types (classes) are the same across all attached MREs, and that (ii) the underlying operating system provides support for virtual memory paging and its manipulation by user-level processes (the MRE in our case).

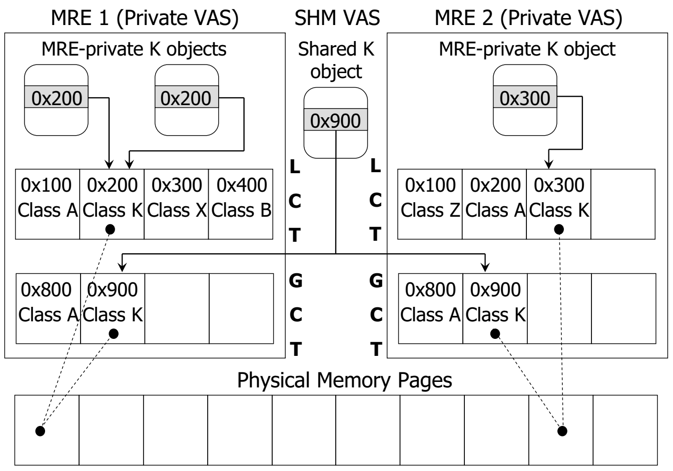


Figure 6.2: Manipulation of VAS mapping so that class pointers resolve to equivalent MRE-private class representations across attached MREs without copying or moving, and while enabling direct retrieval of object metadata (for dynamic dispatch, field access, etc.). Each box is a virtual page (4KB in size), potentially mapped to physical memory. Blank boxes are unmapped. We omit mapping lines (dotted with round ends) for classes other than K, for clarity.

6.2.1 Double Memory Mapping

XMem manipulates the virtual address space to enable direct access to objects as well as to their class representations. Objects in most object-oriented language systems typically contain a reference to an internal representation of the class (type) from which they are instantiated. This reference enables direct retrieval of object metadata for fast implementation of common operations such as dynamic dispatch, static field access, and reflection. These internal representations of classes, however, are MRE-specific and cannot be shared, as they commonly record application- or MRE-specific state and provide access to static (private) data. Class pointers, therefore, must resolve to the MRE-private internal representation of the class.

To avoid moving (reordering) existing class objects (internal representations) within each attached MRE (which can be complex and expensive), yet to ensure that the same virtual address refers to the same MRE-private internal representation of the class in all MREs, XMem aligns class objects to virtual memory page boundaries (we assume traditional 4KB pages) and manipulates virtual address mapping as depicted in Figure 6.2 via double mapping. In the virtual address space (VAS) of each attached MRE in XMem, there is a global class table (GCT) and a local class table (LCT), both of which are MRE-private. The LCT holds the representations of both MRE-private and global classes. LCTs across MREs are independent and unrelated. In contrast, the GCT in each MRE is identical in structure and layout (class order, count) and has the same virtual address in MRE-private space.

XMem maps the physical page of a particular (global) class to a virtual page in both LCT and GCT, to achieve resolution of class pointers within shared objects to private class representations without copying or moving and without introducing pointer indirection. In the example, the class pointer of unshared objects (instances of class K) refers to the internal class representation in the LCT in their MRE (address 0x200 in MRE 1 and 0x300 in MRE 2). When the two MREs share an object of type K, XMem adds an entry for class K to the GCT at the same location in each MRE. Since the GCTs are identical in each MRE and start at the same virtual address, the class pointer in the shared object is the same for both MREs (0x900). We overview the class loading process that makes use of this implementation in Section 6.2.8.

There are two side-effects of this double-mapping. First, in the worst case, XMem consumes twice the VAS needed for classes (worst case is when each MRE-private class is a globally shared class). This case is uncommon in our experience as the number of MRE-private classes typically far exceeds that of globally shared classes. Moreover, such VAS use is negligible for machines with large address spaces (64-bit platforms). Second, class alignment to virtual page boundaries limits the class size to that of a virtual page and can cause fragmentation in the LCT (when classes are smaller than the page size). In practice, we have never found a class object to be larger than our virtual page size. However, if this proves to be a limitation, we can reserve a multiple of the page size for each class. In our implementation, the LCT is the permanent generation of the MRE which stores other long-lived data (e.g. MRE data structures, static strings) in addition to class objects. This data consumes part of each page which helps to reduce fragmentation. We measure and report the space overhead of fragmentation in Java benchmarks in Section 6.3.2. We plan to investigate the impact of large page sizes on XMem systems as part of future work.

6.2.2 Shared-to-Private Pointers

To guarantee that shared objects never refer to private heaps (since such references are particular to a specific MRE process), XMem piggy-backs on the extant write barrier implementation of generational garbage collection (GC). Generational GCs are in widespread use in modern MREs as they provide superior performance which they achieve by exploiting similarity in object lifetimes and by partitioning the heap into distinct, contiguous spaces called generations [155, 166, 99]. These systems allocate most objects from the young generation, and collect this region frequently since a majority of objects die young [16, 98]. To enable efficient, independent collection of generations, generational GCs use a write barrier at every reference store in a program to track references from older to younger generations. Modern MREs typically also employ a permanent generation that is rarely collected and that holds long-lived objects such as internal class representations, constant strings, and MRE data structures.

XMem extends write barriers with two checks needed to compare the source and destination of a particular pointer against the constant boundary between MRE-private and shared part of the heap. We need the source check for each pointer store, and the destination check only for stores to the shared memory. If a program makes an assignment that violates the XMem constraint, the runtime throws an exception and the instruction fails. Since we map the shared memory segment to the same location in each MRE and the segment has a fixed size, this check is very efficient: it consists of a register and constant comparison. Such checks impose negligible overhead on modern architectures because there is no memory access and the branch direction is typically highly biased and thus, easily predictable.

6.2.3 Using XMem

Developers make use of XMem via a simple application programming interface (API). The XMem API for Java comprises the following public static methods declared in the `ipc.SharedMemory` class:

```
void sharedModeOn();   boolean isSharedModeOn();
void sharedModeOff();  boolean isShared(Object o);
Object accept(int p);  void connect(int p, Object o);
void bind(int p);      Object copyToShared(Object o);
```

To support transparency and backward-compatibility, programs within XMem allocate objects using the conventional `new` operator, regardless of whether they are allocating shared or private memory. XMem determines from which region (shared or private) to allocate using a per-thread allocation mode. Initially, the allocation mode is private. Programs or libraries change the allocation mode explicitly via the `sharedModeOn` and `sharedModeOff` methods. The system throws an `ipc.SharedMemoryExce-ption` to prevent shared-to-private pointers as well as signal binding/connection failures and out-of-memory errors.

XMem makes use of the concept of *ports* to enable co-existence of multiple, isolated communication channels in a single shared memory segment. To initiate communication, two distinct MREs (to which we refer as a client and a server) must obtain a reference to a shared object (to which we refer as a root). A client allocates a root in shared memory and passes a reference to it to the `connect` method along with a port to which a specific server has been bound via the `bind` method. The server retrieves the root from the `accept` method. Once the root is exchanged, further communication proceeds according to an application-specific protocol which commonly includes monitor synchronization (wait/notify) on the root. Objects shared through a particular channel are reachable only to threads/MREs that have established the connection. However, an arbitrary number of threads/MREs can share a specific object if a server makes a reference to a shared object available to multiple clients (which use distinct channels for communication with the server).

To enable interoperability with libraries that do not guarantee immutability of the objects they take as arguments, XMem provides a mechanism for recursive (deep) copying of objects into the shared memory via the `copy` method. Object cloning, commonly available from the underlying language (e.g. Java or C#) platform, by default creates shallow object copies and must be overridden on a per-class basis to support deep coping. XMem provides this general service uniformly across classes and applications. XMem uses stack-based, depth-first copying and handles cycles in the object graph by maintaining a hash table that maps the already-visited objects to their copies. We describe how such copying to shared memory interacts with shared-memory garbage collection in Section 6.2.9.

Although, we focus primarily on shared memory, other IPC mechanisms such as signals and message queues can be built on top of XMem in a straightforward way. We have integrated XMem, through the use of its API, into existing communication mechanisms, such as RMI, applying only minimal library modifications. Such XMem-aware implementations provide two paths of execution that the library routine selects based on the proximity of the communication target: one that employs shared memory and one that uses traditional distributed communication.

6.2.4 Dual Mode Object Allocation

To enable cross-MRE object sharing, XMem introduces dual-mode (shared or private) object allocation. XMem extends the common allocation technique of thread-local allocation buffers (TLABs). TLABs are used by modern MREs to reduce contention between threads that concurrently perform linear (bump-pointer) allocation from a common area. This approach requires no synchronization when allocating within a TLAB. The system allocates TLABs to threads linearly, using more expensive atomic operations. XMem associates two TLABs with each application thread, one in private and one in shared memory. We do not initialize the latter until the thread performs its first allocation into shared memory, e.g. when it first executes a `new` bytecode within the XMem shared mode (`sharedModeOn()`). XMem excludes objects that the system creates by side-effect of other operations, such as class loading or lazy data structure initialization, from allocation in shared memory to prevent unintended object leaks. XMem also uses private mode for allocation of all internal data structures (data commonly stored in a permanent area of the heap).

6.2.5 Thread Synchronization

Two locking schemes are commonly used to implement language-level (e.g. Java) monitors in extant MREs: lightweight locking [135] and biased locking [135]. Biased locking optimistically assumes that a single thread uses a monitor (i.e. there is no contention); when this proves not to be the case, biased locking falls back to lightweight locking. Both lightweight and biased locking require a re-design to work with shared memory. XMem adapts and employs lightweight locking since it is the basis for both schemes. We first overview lightweight locking and then describe its implementation in XMem.

Lightweight Locking. To avoid using OS primitives (a pair consisting of a mutex and a condition variable) in the common case of uncontended locking, lightweight locking employs atomic compare-and-swap (CAS) operations. Only when two threads attempt to lock the same object does the MRE inflate the lightweight lock into a heavyweight

OS-backed monitor. Lightweight locking improves performance as user-mode locking is significantly more efficient than system calls.

The MRE stores basic locking information in the object header which occupies one machine word. The lowest two bits encode one of the three possible states: unlocked (UL), lightweight-locked (LL), and heavyweight-locked (HL). When an object is LL (by a `monitorenter` bytecode), the system inserts a lock record into the stack of the thread performing the lock acquisition operation. During stack unwinding (which takes place when an exception is thrown), the system uses lock records to unlock the objects that are locked in the discarded stack frames. Normally, objects are unlocked by a `monitorexit` bytecode generated as part of the epilogue of block-structured critical sections. Each lock record holds a pointer to the locked object and the original value of the overwritten object header.

During locking, a thread attempts an atomic CAS on the object header to replace it with a pointer to the stack-allocated lock record. Lock records are word-aligned, therefore the two lowest bits are always cleared and do not conflict with the locking state bits kept in the header. If the CAS succeeds, the thread owns the monitor. Otherwise, a slow path is taken and the lock is inflated – the object header is CAS-updated to point to a data structure containing a mutex and a condition variable. This data structure is stored in private, MRE-managed, memory.

During the unlock operation of an LL object, a thread tries to CAS-restore the header that it has stored on the stack. On success, no fall-back is needed and the fast path is complete. The CAS failure indicates that the lock was contended for (and inflated) while it was held. Under such circumstances, it is necessary to notify the competing (and now waiting) threads that the object is unlocked. These threads are blocked on the condition variable. When awakened, they unlock the mutex and resume execution by trying to re-acquire the mutex. The mutex and the condition variable are multiplexed here: first they are used to wait until the LL object becomes unlocked and then they are used in a standard way to provide mutual exclusion along with the wait/notify functionality.

Recursive locking in the lightweight case is based on implicit lock ownership – if the object header points into the stack of the current thread then the current thread already owns the lock and in a new lock record on the stack the header field is set to `NULL`. When unlocking, a lock record with the `NULL` header field is ignored. Recursive locking in the heavyweight case uses a counter located in the aforementioned MRE data structure.

Lightweight Locking in XMem. The challenges to lightweight locking in XMem shared memory are three-fold. First, the header of an LL object points into a private thread stack. Such references cannot be interpreted properly across MREs directly. Second, heavyweight data structures allocated in MRE-private memory must now be accessible to other MREs. Finally, POSIX synchronization primitives by default work within a single process.

To address these issues, XMem allocates a lock data structure (LDS) in shared memory, both in case of lightweight and heavyweight locking and uses POSIX object attributes to enable cross-process synchronization. LDS reserves space for a mutex and a conditional variable (which are initialized only in case of inflation). LDS contains a process identifier (PID) and a thread identifier (TID), that together unambiguously identify the owner, as well as a recursion count, the locked object reference, the binary flag used for mutex/condition variable multiplexing, and the original object header. Lock records that are stored on the stack contain only the address of the locked object. An object header, instead of pointing into a stack, always refers to the corresponding LDS, when locked.

XMem maintains an LDS pool in SHM-Meta (metadata area in shared memory). Application threads atomically bulk-allocate multiple LDSes at once from the global pool to reduce synchronization overhead. Each thread holds several LDSes in a local queue with a FIFO discipline. An LDS of an LL object is returned to the thread-local queue when unlocking succeeds (i.e. no contention is detected). An inflated LDS can be freed only during shared memory GC when the HL shared object becomes unreachable.

Invoking wait/notify on an LL object results in the lock inflation. This is necessary as these operations require support from the OS. An important aspect of LL is the hash value computation. MREs typically store the hash value in the object header and lazily initialize it. The hash code, once computed, should never change. Since LL displaces an object header, a race condition arises when an LL object is simultaneously unlocked and its hash code is being initialized. Such circumstances force lock inflation and safe initialization of the hash code (inflated locks are more stable as their unlocking does not change the object header).

XMem uses this modified LL scheme only in the shared memory – each MRE uses the original scheme internally as it is more space-efficient. Each lock/unlock operation checks whether the corresponding object is shared or not and dynamically applies the appropriate locking scheme.

XMem automatically preserves the guarantees provided by the memory consistency model of a specific MRE (e.g. the Java Memory Model [109]) since the system consists of homogeneous MREs.

6.2.6 Global Operations

Each XMem MRE executes a Global Operations (GlobalOp) thread that performs five coordinated *global operations*: attachment, detachment, connection, class loading, and garbage collection (GC). XMem serializes these, relatively rare, operations using a global lock (a mutex and a condition variable located in the shared memory). The system performs every global operation in parallel by all currently attached MREs using the GlobalOp thread in each MRE. Since MRE attachment and detachment are global operations, there is a well-defined set of attached MREs with respect to the current global operation. This is an important property, as global operations terminate only when all attached MREs report operation completion. With the exception of GC, global operations execute concurrently with application threads (i.e. without stopping them).

6.2.7 Attachment, Detachment, and Connection

Two JVM properties, `ipc.shm.file` and `ipc.shm.destroy`, control MRE-OS interaction. The first one identifies a shared memory segment to create or attach to (we employ Linux System V IPC [123]). The second one specifies if an MRE should mark the segment for destruction upon termination. The OS releases only marked segments whose attachment count reaches zero. Upon startup, each MRE attempts to create a new shared memory segment. The creation process fails if the segment already exists, which causes a fall-back to attachment. The MRE that succeeds in segment creation, initializes the shared data structures (located in SHM-Meta).

MREs that attach/detach to/from an existing segment perform a global attach/detach operation. Attachment takes place after completing the MRE bootstrap procedure and before invoking the program's `main` method. Detachment is performed upon program termination. These two global operations are automatic and not accessible via the XMem API. An MRE can attach only to one segment at a time. However, XMem supports multiple communication channels over a single shared memory segment. A configuration with a single segment per host is most memory-efficient but multiple segments can be used if needed. Attach and detach operations update a global counter that tracks the number of attached MREs.

The connection operation establishes a communication channel. Connection allows two MREs to obtain a reference to a shared object while guaranteeing privacy (other MREs cannot reach that shared object). It implements semantics similar to that of a network socket. The arguments passed to the `connect` (i.e. a port number and a shared object), are propagated to other MREs as parameters of the global operation. Each MRE maintains a list of ports to which it is bound. When a connection request to a locally bound port is detected, an MRE adds the corresponding shared object to a local queue and awakens the threads that are blocked on the `accept` call on the port. The shared object is then dequeued and returned by the `accept` method. XMem ensures that only one MRE is bound to any port (an atomically-updated boolean table is kept for this purpose in the shared memory). Since connection is a global operation, it is serialized with respect to GC, and as a result, the shared object (root) has a stable location while the operation is in progress.

6.2.8 Global Class Loading

Through global class loading, XMem ensures that a specific class is privately loaded by, and is the same in, all attached MREs, to guarantee type safety. XMem implements the latter by comparing the 160-bit SHA-1 hash value computed for the class bytecode, across MREs. If an MRE encounters a bytecode mismatch, global class loading fails and an exception is thrown.

Since XMem places no restrictions on MRE-private class loading, the class of a shared object may or may not be loaded in all attached MREs when it is instantiated in shared memory. Therefore, following each object allocation, XMem executes a class loading barrier which checks if the new object resides in the shared memory. If the object is shared, the MRE checks whether its class has been loaded globally. To make this check fast (note that it is done for each allocation), XMem adds a field (a forwarding pointer) to each private class object. The forwarding pointer is initially set to `NULL` to indicate that the class is loaded only privately. After global class loading, the forwarding pointer is set to the GCT address of the class. Following each allocation in shared memory, the MRE updates the class pointer of the new shared object to the forwarding pointer. If the check fails, i.e. the class of the new shared object has not been loaded globally, the MRE initiates global class loading.

Global class loading uses the default system class loader (which corresponds to the `CLASSPATH` variable). XMem permits classes defined by user-defined class loaders to be instantiated in the shared memory as long as the corresponding user-defined class loaders are themselves allocated in the shared memory. However, even though class loaders can be shared, the internal class representations are always MRE-private. XMem relies on MRE-private class loader constraints to guarantee type safety in the presence of lazy class loading, user-defined class loaders, and delegation [104]. No extension is needed because we first locally load all globally loaded classes and thus, local constraints are always a superset of global constraints.

6.2.9 Global Garbage Collection

Global GC in XMem identifies and reclaims dead, shared objects (i.e. those that are not reachable from any attached MRE). The GC is initiated by one of the attached MREs when allocation of a new TLAB in shared memory fails. In order to interoperate with different GC algorithms and heap layouts [167, 99], XMem provides a generic mechanism for identifying root objects in the shared memory. Root objects in this context are objects directly reachable from one or more MREs by following pointers that are located on thread stacks, in registers, or in the live part of a private heap. Once a snapshot of the root objects is obtained, shared memory can be collected in a conventional way using any tracing collector.

The key challenge is in identifying the root objects without resorting to scanning all the live objects in each MRE. Note that pointers into shared memory can be scattered across all generations. At the same time, we can expect the number of such pointers to be relatively small.

XMem identifies roots by piggy-backing on a fast minor collection (the one confined to the young generation). To enable this, XMem extends a card table mechanism [169] that supports generational GC so that it tracks pointers from older generations that point into the young generation or into shared memory. As a result, a young generation collection is able to detect all root objects that originate from a given MRE without an exhaustive scan of older generations. For each global GC, XMem triggers a minor collection in the attached MREs. To perform a minor GC, state-of-the-art MREs typically employ a parallel copying collector [67] that is executed in a stop-the-world (STW) fashion as it imposes very short pause times (i.e. concurrent collection [57, 122] is not necessary).

An XMem system implements global GC of the shared memory segment using STW parallel copying collection. All attached MREs perform GC in parallel, each contributing multiple GC threads. MREs synchronize only before and after collection. A full barrier is needed after all MREs reach a safepoint (i.e. state where application threads are suspended) because one cannot start moving the shared objects while other MREs are actively using them. For similar reasons, all GC threads from all MREs synchronize when leaving a safepoint. Any additional coordination depends on the GC algorithm used. Although, global GC stops all MREs, it is not unscalable or deadlockprone since bringing an MRE to a safepoint is a low-delay operation robust with regard to I/O.

Since global GC can interrupt an XMem deep copy from private to shared memory, we must be careful to avoid introducing temporary shared-to-private pointers during the copy process. To this end, when we copy an object to its new location, we clear its reference fields (as they may still point to private objects). We update these fields with the correct values (new locations) when we copy the corresponding objects to shared memory. Global GC needs to update the entries in the stacks and hash tables used by XMem copy operation because it is moving objects. We provide a new object header to each shared memory replica to preclude them from inheriting the synchronization state of original objects.

Non-global GCs (both minor and major) do not follow pointers that point into the shared memory. Because of the XMem invariant that no shared-to-private pointers are allowed, it is correct to stop tracing when a shared object is encountered. GC completeness is preserved because scanning of objects in the shared memory cannot lead to the discovery of any additional live objects in the private heap. Local GC performance, is thus the same regardless of the number of objects in shared memory.

The most suitable GC algorithm for shared memory collection depends on the demographics and total size of the live shared objects [98]. If XMem is used to share a large amount of long-lived data, then compacting collectors are most appropriate. On

Bench-mark	Gen. Size [MB] Young+Old	Gen. Size [MB] Perm.	Execution Time [s]	GC Count Minor	GC Count Major	Class Count
bloat	40	5	55.3 ± 0.5	528 ± 2	1 ± 0	827
pmd	34	6	19.5 ± 0.1	495 ± 1	8 ± 1	1186
xalan	42	6	49.2 ± 0.3	1480 ± 8	107 ± 4	1179
antlr	8	4	4.5 ± 0.1	380 ± 1	5 ± 0	679
chart	30	9	15.7 ± 0.1	355 ± 7	9 ± 0	1440
eclipse	68	16	66.5 ± 0.2	551 ± 3	11 ± 0	2627
hsqldb	336	5	13.6 ± 0.1	9 ± 0	5 ± 0	736
fop	20	6	2.0 ± 0.0	19 ± 0	0 ± 0	1423
luindex	8	4	7.2 ± 0.1	260 ± 8	3 ± 0	689
lusearch	18	4	8.6 ± 0.1	706 ± 1	1 ± 0	683
jython	8	8	43.1 ± 0.3	3539 ± 2	1 ± 0	1325
jbb/6wh	476	8	90 ± 0.0	149 ± 0	4 ± 0	1296
jbb/8wh	636	8	90 ± 0.0	116 ± 1	3 ± 0	1296
jbb/10wh	780	8	90 ± 0.0	98 ± 0	3 ± 0	1296

Table 6.1: Java benchmarks that we use to evaluate XMem overhead. For each benchmark, we report generation sizes, execution time (note that Jbb runs for a fixed period of time), the number of minor/major collections, and the number of loaded classes.

the other hand, if the primary purpose of XMem is communication between strongly isolated MREs, then copying collection is a better choice [166]. This is because the communicating MREs exchange a small number of objects which exhibit relatively short lifetimes. Generational collection can be used to support a wide range of object lifetimes. To accommodate short-lived communication behavior typical of J2EE applications, we implement a non-generational, parallel copying in our XMem prototype.

Parallel copying collectors employ several GC threads to evacuate live objects from the currently-used source space(s) to the currently-unused target space [85]. Since most objects are expected to be unreachable, the target space is typically smaller than the

Bench-mark	XMem Overhead	
	Time [%]	Space [MB]
bloat	3.5	2.24
pmd	3.5	2.94
xalan	3.4	3.11
antlr	2.8	1.79
chart	1.9	3.74
eclipse	2.3	7.04
hsqldb	0.3	2.00
fop	2.8	3.45
luindex	1.6	1.83
lusearch	2.6	1.79
jython	3.0	3.14
jbb/6wh	0.64	3.59
jbb/8wh	1.78	3.59
jbb/10wh	0.82	3.59

Table 6.2: The overhead introduced by XMem in terms of application throughput (Jbb) or execution time (DaCapo) and occupancy of the permanent generation.

source space(s) and the worst-case scenario is handled by falling back to the promotion of overflow objects into older generation(s). In the absence of a generational heap layout, half of the space needs to be set aside as a copy reserve.

XMem employs two equal-sized semi-spaces in the shared memory and the collection of the source semi-space is performed in parallel by all attached MREs. This process is interleaved with local minor GCs so that the object graph is traversed only once. Each MRE uses multiple GC threads, which correspond to schedulable kernel threads and whose total number equals the number of processors/cores available or dedicated to each MRE.

XMem employs a two-level load balancing scheme in the form of work stealing [67]. GC threads that become idle attempt to steal object references from non-empty marking stacks of other GC threads. Each GC thread is associated with two marking stacks, which we refer to as the local and shared stack. Intra-MRE load balancing is limited to local stacks while inter-MRE work stealing uses shared stacks only. MREs push references to objects residing in the shared memory onto the shared stacks to make them available to other MREs. Local load balancing is preferred and global stealing is done only when all local stacks become empty. The stealing target (i.e. the marking stack/stack entry) is selected randomly.

Global GC is an STW operation that comprises three barriers: prologue, epilogue, and GC termination. The GC prologue flips the semi-spaces. The GC epilogue forwards the pointers in SHM-Meta and deflates heavyweight monitors associated with dead objects.

To ensure that each live object is processed exactly once, GC threads claim objects atomically. Atomic CAS instructions are supported by most processors and can be used across processes (as they are based on physical rather than virtual addresses). To reduce contention, each GC thread owns a parallel local allocation buffer (PLAB) where it copies the objects it has claimed. We allocate PLABs linearly, atomically, and on-demand, from the target semi-space. The GC first copies an object to its destination, and then pushes the addresses of its reference-type fields onto the marking stack (local and/or shared). Then, a GC thread tries to CAS-forward the original object header to the new location. If a thread loses a race to another thread, the GC removes the object from the PLAB and pops the new pointers off the stack. This order of operations is motivated by fault-tolerance (Section 6.2.11).

6.2.10 Global Meta-Data Management

The SHM-Meta data structures support the runtime and global operations of XMem. They include a descriptor for the current global operation, which contains the operation code, its input arguments, barrier counters, state flags, and a mutex and condition variable with which the system serializes the execution of global operations. In addition, SHM-Meta holds the marking stacks for global GC and a table that records the meta-information for all globally loaded classes including the class name, defining class loader (set to `NULL` if the default system class loader is used), and a bytecode hash for type-safety verification. SHM-Meta also holds a list of the bound ports that are currently in use for communication sessions between co-located MREs. Finally, SHM-Meta contains single-word entries for (i) the number of attached MREs, (ii) the number of globally loaded classes, (iii) the boundaries of and current position in the shared heap (for allocation of new TLABs), and (iv) the start and end of a pool of global locks that enable cross-MRE monitor synchronization.

6.2.11 Fault Tolerance

XMem tolerates unexpected MRE termination, between and during global operations, by implementing a timeout mechanism (based on the `pthread` timed wait on a condition). If an MRE fails, the next global operation times out. Upon timeout, XMem subtracts the number of not-responding MREs from the counter of the attached MREs and releases any shared locks that were held by the terminated MRE. Connection, and class loading are global operations that do not require any additional handling upon timeout.

In case of timed-out detachment and attachment operations, the system needs to determine whether it was the detaching/attaching MRE that failed (to correctly keep track of the number of live MREs). This is done based on the PID of the process which initiated attachment/detachment (XMem sends a signal using the `kill` system call and gets an error if the process is dead).

GC requires more complex handling, as the shared stacks of a terminated MRE can contain pointers stolen from other MREs. These stacks are located in shared memory so they are not lost and can still be processed. During GC, objects are forwarded to their new locations only when they have been copied and when their content has been scanned (and pushed onto a stack). Therefore, copying collection can be interrupted at any time without losing correctness, provided that whatever is on the stack(s) is eventually processed. If a global GC times out, it is sufficient to empty all the marking stacks located in the shared memory.

6.2.12 Implementation Details

We have implemented XMem in HotSpot [118], an open-source, high-performance JVM written in C/C++. The heap in HotSpot [85] comprises three generations: young (where new object allocations take place), old (where long-lived objects are promoted), and permanent (where classes are stored). HotSpot reserves two words per object. The first word (the header) contains the locking state, age bits, and the hash code. The second word is a pointer to a class object located in the permanent generation. Class objects encapsulate static fields, a virtual method table, a class loader reference, and pointers to other meta-objects that describe methods and fields (among others).

The `PTHREAD_PROCESS_SHARED` attribute is set on the POSIX mutexes and condition variables to enable cross-process synchronization. To create or look up a shared memory segment, XMem employs `shmget`. This system call is used with the `IPC_PRIVATE` key to implement double mapping in LCT and GCT. Global shared memory segments are identified by a key generated by `ftok` based on a file name. XMem supports multiple global segments on a single host, differentiated by a file name (the JVM `ipc.shm.file` property). We implement attachment with `shmat`, which allows to specify a virtual address that a segment is mapped to. MRE-private memory is allocated using `mmap`, which is called with the `MAP_FIXED` flag when pinning GCT at a specific location. For atomic operations we use the x86 `cmpxchg` instruction. LCT corresponds to the permanent generation.

6.3 Experimental Evaluation

We evaluate time/space overhead imposed by XMem extensions by comparing the performance of standard Java benchmarks run on top of an unmodified HotSpot JVM and XMem-enabled one. To measure the impact of XMem on communication performance (throughput and latency) we use a set of microbenchmarks for common Java RPC protocols. We also investigate XMem impact on end-to-end application performance.

6.3.1 Methodology

Our experimental platform is a dedicated machine with a dual-core Intel Core 2 Duo (Conroe B2) processor clocked at 2.66GHz, equipped with 4M 16-way L3 cache, 32K 8-way L1 cache, 2GB main memory, and running Debian GNU/Linux 3.0 with the 2.6.17 kernel. We use the HotSpot OpenJDK [118] v7-ea-b18 (Aug. 2007) compiled with GCC 3.2.3 in the optimized client-compiler (C1) mode. This version of HotSpot implements a highly-optimized, state-of-the-art serialization mechanism and uses standard (not process-shared) mutexes/condition variables.

For our experiments, we employ standard community benchmarks from the DaCapo [54] and SPECjbb2005 [145] suites to evaluate the impact of XMem on programs that do not communicate across MREs. We use the large input for DaCapo and 6, 8, and 10 warehouses, with 90s runs, for Jbb. To evaluate the impact of using shared memory, we develop a number of benchmarks ourselves (an approach taken in [107] in a similar context), which exercise shared memory and implement the J2EE communication protocols. We describe these benchmarks with each experiment. We evaluate XMem-aware implementations of RMI and CORBA, serialization and XML, JNDI, and TCP/IP sockets. Finally, we evaluate XMem for two server-side applications: Hsqldb [86] and Tomcat [3]. In all experiments, there are 2 MREs and the shared memory size is 30MB. Whenever running the original HotSpot JVM, we set the young generation to 30MB.

6.3.2 XMem Overhead

To investigate the overhead introduced by adding support for XMem, we compare the performance of shared-memory-oblivious applications run on top of an unmodified HotSpot JVM against our implementation of XMem. In Table 6.1 we present basic statistics for the benchmarks that we use (we report generation sizes, execution times, the number of major/minor collections, and the number of loaded classes). Table 6.2

summarizes the results. For each benchmark, we employ a heap size (i.e. total size of the young and old generation) of twice the minimum. We employ this methodology [146] to ensure some GC activity without having GC dominate performance – so that we are able to measure other sources of overhead potentially introduced by XMem. We set the permanent generation size to the minimum required for XMem to load all the necessary classes. The young generation constitutes one fourth of the heap. We report generation sizes, the number of minor and major GCs, and the number of loaded classes. For timings, we execute 5 warm-up runs then compute the average and standard deviation of the next 5 runs.

XMem imposes negligible time overhead which we express as the percentage of total execution time (for DaCapo) or throughput (for SPECjbb2005). The sources of overhead are two additional checks per write barrier and internal checks for whether or not an object is shared. We report absolute values for the space overhead introduced in the permanent generation (by the page alignment implementation) as this overhead does not depend on generation sizes (only on the number of loaded classes). The space overhead ranges from 1MB to 7MB and on average is 3.1MB across the 14 programs. This overhead is bounded by the meta-data size (as opposed to the application working set size).

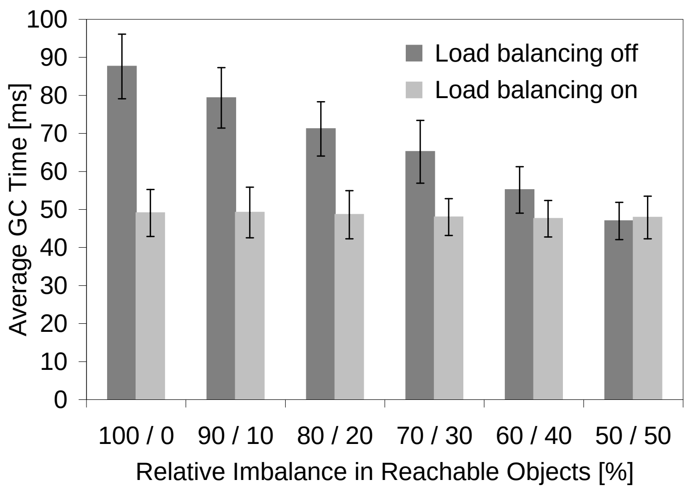


Figure 6.3: Global GC pause times with and without inter-MRE load balancing for different distributions (percentage) of shared objects reachable from individual MREs. The distribution of reachable objects (imbalance) is shown as a pair of percentage values, e.g. 90/10 means that 90% of objects are reachable from one MRE and the remaining 10% from the other.

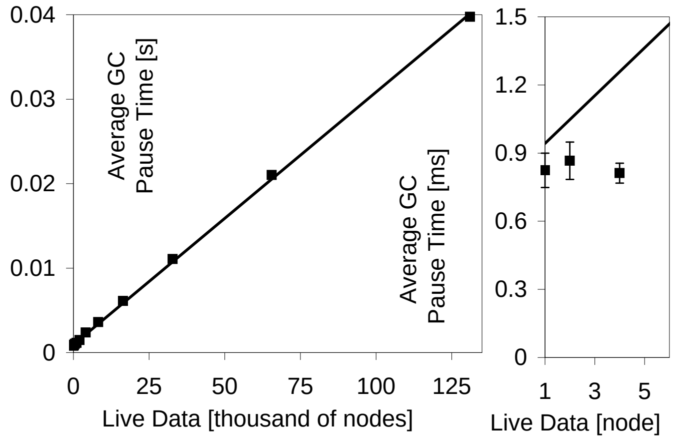


Figure 6.4: Impact of the size of live shared objects on global GC pause times. We present two views of the same graph to show both throughput and latency of STW global GC. We use regression to obtain the parameters of the linear relationship between live data size and GC pause time.

6.3.3 Global GC Performance

Figure 6.3 shows the impact of inter-MRE GC load balancing (work stealing) on average pause times of global GC. In this experiment, each MRE executes a single GC thread and can reach only a specific fraction of shared objects. We express the distribution of reachable objects (imbalance) as a pair of percentage values. For perfect balance (50/50), load balancing adds a small overhead. For the most imbalanced configuration (100/0), inter-MRE work stealing reduces GC pause time by 44%. We report average GC pause times (and standard errors) from 15 GCs. This result indicates that cross-MRE load balancing is important for efficient GC in an XMem system.

XMem implements STW parallel copying collection and therefore its GC pause times increase linearly with live data size. Figure 6.4 presents measurements obtained using two MREs, each with a single GC thread, where live data consists of a binary tree comprising a specific number of nodes. We report average GC pause times for different live data sizes. Global GC latency (computed by extrapolating GC pause time for live data size equal to zero) is 0.9ms. Safepoint latency in a single MRE is 0.7ms on average. Safepoints are reached concurrently by two MREs (they do not add up). Thus, there is 0.2ms overhead imposed by XMem to coordinate global GC across MREs. Copying throughput is 3.3 million nodes/second (where each node corresponds to 5 small objects). This throughput is identical in case of a single MRE (XMem does not degrade it).

6.3.4 Communication Efficiency for Microbenchmarks

We next evaluate the impact of XMem on the performance of Java communication technologies using our microbenchmarks.

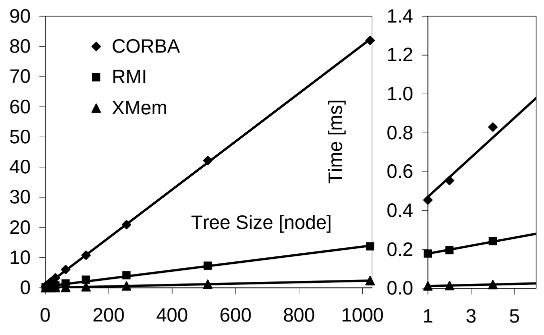
RMI and CORBA. RMI [131] enables inter-MRE type-safe remote method calls. A server registers a remote object using a directory service which is later consulted by the client to look up the remote object by name. Once a remote reference (proxy) is constructed, the client can call remote methods. A client and a server use automatically-generated stubs and skeletons to (de)marshall arguments and return values. CORBA [50] employs a more portable transport protocol (IIOP) to interoperate with other runtimes. Our microbenchmark times a remote method call that takes a binary tree of objects as an input argument and returns another binary tree as an output value. We employ binary trees as the microbenchmark since they represent a middle-ground in common data structures: they are neither sparsely-connected (like linked lists) nor densely-connected (like complex graphs).

Figure 6.5(a) shows the average invocation time (y-axis) for an increasing number of nodes in the binary tree (x-axis). We implement the remote call using XMem by allocating binary trees directly in the shared memory. Client-server interaction is coordinated by monitor synchronization. Having allocated a tree, the client notifies the server that the input is ready. Once the server allocates the output tree, the client is notified that the call is complete. XMem reduces latency 15x and 37x while increasing throughput (calls/second) 6x and 35x, compared to RMI and CORBA, respectively, since XMem avoids argument marshalling and network communication.

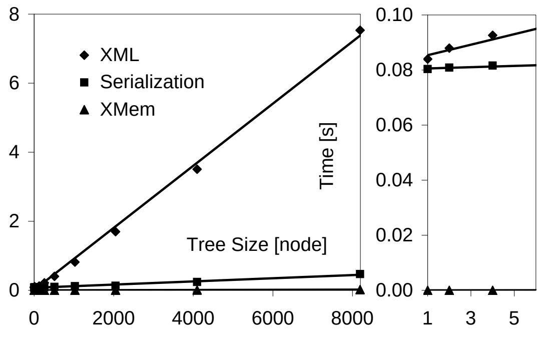
Serialization and XML. Object serialization [141] provides a type-safe mechanism for transforming an arbitrary graph of objects implementing the `java.io.Serializable` interface into a binary byte stream which then can be used to reconstruct the original data structure. A runtime-portable alternative to binary representation is XML. We compare default and XML-based serialization against their XMem implementation. Our microbenchmark times the exchange of an object graph between a server and a client. A client allocates a binary tree of objects, serializes it and sends the result to the server over a socket. The server deserializes the tree, allocates a response (being a binary tree of the same size) and sends it back to the client in a serialized form. In XMem, we allocate the tree in the shared memory and notify the other side that the data is ready (we consider the overhead of copying below). Figure 6.5(b) presents the average serialization time (in msecs on y-axis) for a tree of 1–1024 nodes (x-axis). XMem eliminates the need for serialization and data transfer and thus improves throughput (calls/second) 20x and 391x while reducing latency by around 7000x compared to default and XML serialization.

JNDI. JNDI provides access to directory services, such as LDAP or RMI registry, where clients can look up objects by name as well as evaluate search queries. Our microbenchmark first binds a specific number of objects in an RMI registry and then

(a) Remote method invocation time (ms) for binary tree pass/return (x-axis is node count).



(b) Object serialization time (s) for client/server binary tree send/receive (x-axis is node count).



(c) Object lookup time when a directory server returns a number (x-axis) of name/object pairs (bindings).

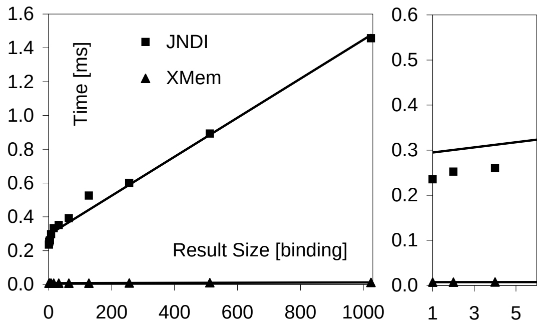


Figure 6.5: Microbenchmark communication performance. We blow up the axes using a second graph snapshot to make latency visible. Each graph shows regression lines.

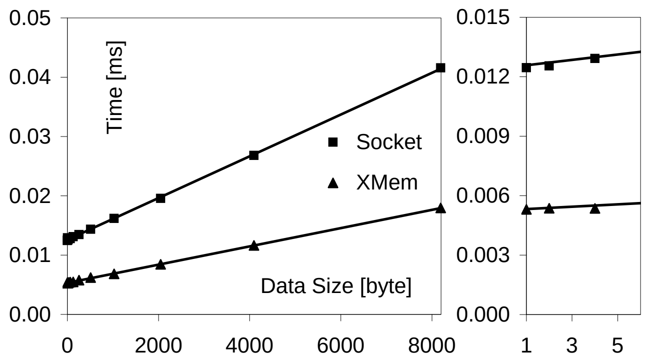


Figure 6.6: Data transfer time (ms) for client/server array send/receive (x-axis is array size in bytes).

performs a query that lists all available bindings (name/object pairs). We time the latter operation only as it is more important (directories are rarely modified). XMem keeps the bindings in the shared memory and returns an enumeration of their subset in response to each query. This enumeration is allocated in the shared memory and returned to the client by means of a notification. Figure 6.5(c) shows the average results gathered for a varying number of bindings (1–1024). XMem reduces latency 32x and increases throughput (lookups/second) 240x which can be attributed to copy and transfer avoidance.

TCP/IP Sockets. Network sockets operate at the byte level (as opposed to the object level) and therefore have no notion of type-safety. However, we compare their efficacy with XMem for completeness. Our microbenchmark measures the time needed to transfer a byte array of a certain length from a client to a sever and vice versa using TCP/IP sockets. We implement XMem-based communication by allocating a shared byte buffer. Each party writes into the shared buffer and then notifies its peer that the new data is available. Figure 6.6 compares the transfer time (in ms) using conventional sockets for data sizes 1 to 8192 bytes (x-axis). XMem increases throughput and decreases latency both by 2x by avoiding network stack interposition and redundant data copying.

Copying Overhead. Occasionally, an object graph needs to be copied to the shared memory to ensure full transparency of communication. In case of remote method invocation and object serialization this translates to allocating locally and then copying an object tree to the shared memory just before notification. In case of sockets, two copies are necessary in the worst case: first from a local buffer (client side) to a shared buffer and then from the shared buffer to a local buffer (server side). Since bindings in directory services are immutable, it is sufficient to copy only the enumeration object encapsulating the query result while leaving the bindings intact. Table 6.3 shows the impact of copying on latency and throughput. We report relative performance of XMem with copying to existing technologies run on top of HotSpot (HS) and the non-copying version of XMem. Columns 2 and 4 show latency and throughput for XMem with copying vs. HotSpot and columns 3 and 5 show these metrics for XMem without copying vs. XMem with copying. When copying is used, XMem still significantly outperforms the extant mechanisms (Columns 2 and 4).

Bench-	Latency		Throughput	
mark	vs. HS	vs. XMem	vs. HS	vs. XMem
RMI	2.5x	5.8x	2.4x	2.4x
CORBA	6.4x	5.8x	14.3x	2.4x
Serial.	1152x	6.1x	8.3x	2.4x
XML	1204x	6.1x	164x	2.4x
JNDI	6.9x	4.6x	71x	3.4x
Socket	2.2x	1.1x	1.5x	1.6x

Table 6.3: Impact of copying shared data on latency and throughput. Columns 2 and 4 show these metrics for XMem with copying vs. HotSpot and columns 3 and 5 show these metrics for XMem without copying vs. XMem with copying.

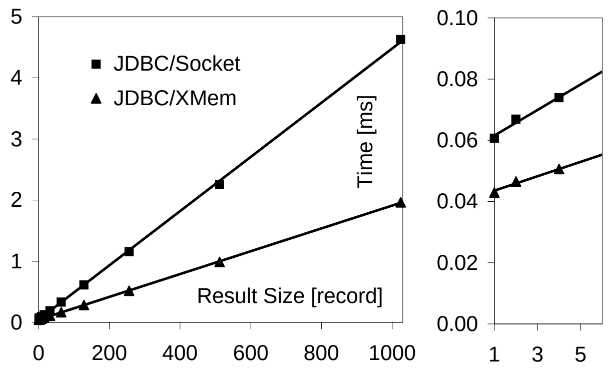


Figure 6.7: Application performance: Hsqldb. We report database query processing time (ms) when a server returns a set of records (x-axis is number of records).

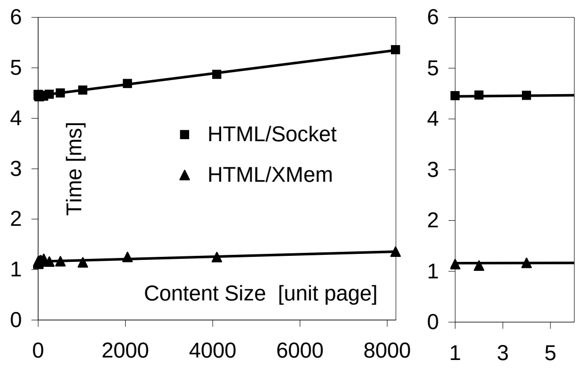


Figure 6.8: Application performance: Tomcat. We report request processing time (ms) when a web server retrieves a web page (x-axis is page size).

6.3.5 Application Performance

We next evaluate the impact of XMem on the performance of two enterprise applications. We quantify the improvement in user-perceived throughput and response time by comparing an unmodified database server (Hsqldb) and a web server (Tomcat) with their XMem-based variants.

Hsqldb [86] is a relational SQL database management system that supports in-memory and disk-based data storage. JBoss uses an embedded Hsqldb database engine by default for persistence and caching. We have modified Hsqldb 1.8.0 to employ shared memory. A client allocates an SQL query as a shared string. The server is then notified, parses the query, and computes the result in the shared memory. Hsqldb maintains an object cache in the shared memory. Internal representation of leaf data in the object cache is based on immutable objects (strings, integers, dates that model SQL objects). Clients can be given a reference to such objects without a risk of modification and therefore most data (and metadata) does not need copying. We have encapsulated interaction over XMem into a JDBC driver for Hsqldb to achieve full transparency. The server listens for connections both on a network socket and in the shared memory. For Hsqldb we measure the impact of XMem on end-to-end throughput (queries/second). Our microbenchmark times the `SELECT * FROM T` statement executed against a table `T` which contains between 1 and 1024 3-field records. Figure 6.7 and Figure 6.8 show the results. XMem increases throughput 2.3x and decreases latency 1.4x. The

Hsqldb JDBC driver performs proprietary data serialization, which is unnecessary in XMem.

Apache Tomcat [3] is an industrial-strength web and servlet container. We have modified Tomcat 6.0 to optimize local request handling using XMem. A client and a server share a byte array and notify each other when sending data. We measure end-to-end performance (requests/second) when retrieving (HTTP `GET` method) static web pages of different sizes (multiples of a unit page size). We use the Apache `httpclient` package to generate conventional HTTP requests. Figure 6.8 shows the time needed to retrieve a page of a given size. XMem achieves 4x better throughput and 4x shorter latency.

6.3.6 Results Summary

Table 6.4 summarizes our results in terms of average latency and throughput. We use least-squares linear regression to obtain latency and throughput as the coefficients in the equation $time = latency + size/throughput$, following [37]. We report throughput in the units appropriate for each protocol. While microbenchmarks focus on communication efficiency (the only additional processing is initialization/allocation of the exchanged data), Hsqldb and Tomcat provide insight into the end-to-end application performance. We observe very significant reduction in latency (over three orders of magnitude) in case of serialization (default and XML-based) – RMI, CORBA, and

Bench-	Latency		Throughput	
mark	HS	XMem	HS	XMem
RMI	0.18 ms	15x	75.8 call/s	6x
CORBA	0.45 ms	37x	12.5 call/s	35x
Serial.	80.4 ms	6977x	21.8 object/s	20x
XML	84.0 ms	7292x	1.11 object/s	391x
JNDI	0.24 ms	32x	833 lookup/s	240x
Socket	0.01 ms	2.3x	279 kB/s	2.3x
Hsqldb	0.06 ms	1.4x	227 query/s	2.3x
Tomcat	4.46 ms	3.9x	10^4 request/s	4.2x

Table 6.4: Summary of XMem impact on latency and throughput for microbenchmarks and applications. We report average baseline performance (Columns 2 and 4) and XMem improvement as a multiple of the baseline (Columns 3 and 5).

JNDI use their own, more efficient, serialization and thus benefit less due to XMem. XML-based serialization yields the most significant throughput increase (over two orders of magnitude) since it uses a verbose representation of the object graph and thus transfers more data.

6.4 Related Work

The key difference between XMem and previously reported systems that coordinate co-located and isolated applications written in type-safe languages is that XMem takes a top-down approach by assuming full isolation between MREs and providing an efficient and straightforward mechanism for direct object sharing while preserving strong OS-assisted resource protection as much as possible. Prior work has focused on bottom-up approaches by introducing weak isolation implemented through replication of basic

OS facilities within a single OS process. Such systems are much more complex than XMem, have weaker protection guarantees, and duplicate existing OS mechanisms.

KaffeOS [10], the Multi-tasking Virtual Machine (MVM) [53], and MontyVM [115] employ a single-application MRE and add support for isolation and multi-tasking. MVM provides isolation via the Isolate API [92]. Multiple programs (tasks) execute in a single MRE instance (OS process) and the MRE manages resources and sharing across them. MVM introduces a level of indirection when accessing static fields and does not support direct object sharing. The system introduces links (communication channels between tasks) but cannot eliminate the object serialization and data copying. KaffeOS supports direct object sharing by means of shared heaps. However, shared heaps are not garbage collected and are coarse-grained entities reclaimed in full when they become unreferenced. KaffeOS lacks support for many state-of-the-art MRE mechanisms like parallel GC and modern synchronization.

Other systems that implement the process/task model within a JVM, include Alta [11], GVM [11], and J-Kernel [157], as well as a multi-tasking JVM described in [18]. These systems strive to provide resource management and isolation within a single process without relying on hardware/OS protection. Class-loader-based isolation [52] is a standard technique commonly employed by applications servers in order to avoid name space pollution/conflicts between multiple web applications hosted within a single JVM. Such isolation, however, does not prevent interference through static fields of classes loaded by the system (bootstrap) class loader. This last problem was addressed in [38] by introducing a control access model called object spaces where cross-space object accesses are mediated by a security policy. This approach, however, provides weak isolation and imposes overhead on inter-space method calls.

XMem does not have the aforementioned limitations and is significantly simpler than multi-tasking approaches as it leverages the existing infrastructure both at the MRE and OS level. At the same time, XMem offers better fault containment – critical errors do not automatically propagate to other MREs unless a fault affects the shared memory. This decreases the probability of a failure escalating to multiple components. In XMem, MREs are not completely isolated as they share part of their virtual address spaces. However, XMem is significantly more robust than multi-tasking approaches, given that resources other than memory are fully isolated and memory itself is only partially shared. XMem achieves stronger isolation, while providing direct object sharing without introducing any level of indirection (unlike the MVM).

The notion of transparent global and local objects in the context of distributed shared memory (DSM) multi-processors has been used in Split-C [51] and UPC [64]. Unlike XMem, these systems are not type-safe and provide access to global objects at a different cost than to local objects. JavaSpaces [68] provide DSM for applications that implement object flows. Object repositories in JavaSpaces are type-safe but the system uses serialization and provides no shared memory support for co-located application components. Other DSM systems for type-safe languages include single-system-image approaches to implementing a global object space such as cJVM [6], JAVA/DSM [178], JESSICA [106], Hyperion [108], JavaParty [124], and MultiJav [41]. While XMem targets sharing between co-located MREs, software DSM focuses mostly on distributed protocols necessary to guarantee memory consistency and cache coherence models defining certain semantics for concurrency in a distributed system.

Runtime systems for concurrent languages that offer built-in constructs for inter-process communication include Erlang [8], Occam [117], and Limbo [63]. These systems build on the algebra of communicating sequential processes [81] and provide a point-to-point message passing mechanism for lightweight processes with share-nothing semantics. In contrast, XMem adheres to the shared memory programming model. Unlike XMem, Erlang is a functional language and requires the shared objects to be immutable. XMem targets general-purpose imperative procedural languages.

In language-based operating systems [138], such as Singularity [65, 89], JX [74], JNode [96], Inferno [63], SPIN [27], Oberon [170], and JavaOS [94], processes share a single address space and use type and control safety provided by a trusted compiler (via static analysis) to guarantee memory protection and resource isolation without implementing a hardware-assisted reference monitor. Singularity is a micro-kernel OS, implemented mostly in C#, supporting efficient communication between multiple isolated processes. Its design differs from XMem in several ways. First, XMem leverages hardware memory protection, while Singularity provides lightweight software-based isolation via type-safety (multiple applications execute in a single address space). Second, Singularity provides message-passing via typed channels and explicit communication primitives. In contrast, XMem provides a shared-memory-based implicit communication where only the initial handshake employs the channel abstraction. Next, in Singularity communication is limited to two endpoints and involves the transfer of ownership of a memory block (there is no data sharing between the sender and the receiver). XMem enables direct and transparent object sharing between any number of threads, potentially from distinct MREs. Finally, Singularity employs block-based reference counting garbage collection while XMem uses more fine-grained tracing GC.

To date, virtual memory manipulation (which is used by XMem to implement double mapping of the GCT and LCT) has been used in MREs mostly in the context of GC [159, 101, 175, 80, 46]. For example, the Compressor [101] employs double mapping to enable concurrent compaction, and the Mapping Collector [159] compacts free space by remapping to avoid object copying.

6.5 Summary and Conclusions

XMem provides type-safe and transparent shared memory for isolated, co-located MREs that implement the same language. The motivation behind XMem is more efficient, cross-component interaction and communication in enterprise multi-tier applications deployed on a single host. XMem provides stronger fault and resource isolation than previously reported systems, while enabling efficient direct object sharing over private channels. To guarantee type-safety, XMem extends state-of-the-art MRE services such as synchronization, class loading, object allocation, and garbage collection, as well as introduces global operations to coordinate MREs using a single shared segment. XMem manipulates virtual memory mapping (via a standard OS interface) to avoid indirect memory access. XMem is transparently integrated within the MRE infrastructure and can be used to optimize existing communication protocols, such as RMI. We implement XMem in the HotSpot JVM and evaluate it empirically. XMem introduces tolerable space/time overhead while improving efficiency (latency and throughput) of extant J2SE/J2EE communication mechanisms by up to several orders of magnitude.

The text of this chapter is in part a reprint of the material as it appears in [160].

Chapter 7

Type-Safe Sharing for Heterogeneous Runtimes: Improving Cross-Runtime Memory Management Performance and Programming Model Using Shared Memory

In this chapter, we describe another approach to improving cross-runtime memory management by using OS support for shared memory. However, unlike in Chapter 6, which focuses on homogeneous (Java) runtimes, this time we investigate object sharing between heterogeneous runtimes, for both static and dynamic programming languages. Specifically, we discuss the design and implementation of type-safe, transparent object sharing in a multi-runtime, multi-language system deployed on a shared-memory multi-core or multi-processor architecture. We overview the design tradeoffs involved in reconciling the major differences in object models, memory models, type systems, and core libraries across languages. We describe in detail the mechanisms needed for an efficient implementation of object sharing in the multi-language setting, including type mapping, class versioning, a language-neutral object/memory model, lightweight monitor synchronization, and a loosely-coupled on-the-fly collector. In addition, we investigate how to incorporate shared memory support into an unmanaged language (we focus on C++) while still guaranteeing type- and memory-safety for the shared objects. Finally, we empirically validate the key benefits of cross-language object sharing: improved communication performance (by avoiding expensive object marshalling) and a richer programming model (by replacing RPC-style interaction with transparent shared memory).

7.1 Introduction and Motivation

Large, scalable software systems are increasingly being built using collections of components to better manage software complexity through reusability, modularity, and fault isolation. Since each programming language has its own unique combination of performance, speed of development, and library support, different software components are often implemented in different languages. As evidence of this, Thrift [143] and Protocol Buffers [129] have been developed by engineers at Facebook and Google, respectively, to enable more efficient interoperation across multi-language components employed within their applications and backend services. For web applications, different languages are better suited for the implementation of different tiers: Ruby, Python, Java, and JavaScript facilitate fast development of the presentation layer, Java, PHP, Perl, Python, and Ruby components commonly implement server-side logic, and Java, query languages, and C/C++ are used for a wide range of backend database technologies. The components of these multi-language, multi-component applications and mashups typically execute within independent runtime systems (language virtual machines (VMs), interpreters, etc.) and communicate and interoperate via remote procedure calls (RPC) and message passing.

Increasingly, administrators co-locate runtimes to better utilize multi-core resources. This makes it possible to use shared memory for such cross-component communication as well as for a cross-runtime language-neutral transparent object storage. However, despite its growing practical value, shared memory has not yet been investigated in either of these contexts. To evaluate the potential of using shared memory for cross-language, safe, transparent communication and object storage, we design and implement *Co-Located Runtime Sharing (CoLoRS)*. CoLoRS provides direct object sharing across static and dynamic, object-oriented (OO) languages.

CoLoRS virtualizes VM components that assume a language-specific object/class/-memory model. In CoLoRS, shared objects retain their language-specific behavior, including the semantics of virtual method calls, locking, and field access. In addition, builtin/library data structures, such as collections, transparently map to their shared counterparts in the CoLoRS object model. Our key hypothesis is that sharing objects across static/dynamic OO languages using shared memory can be safe, transparent, and efficient.

CoLoRS defines an object model and memory model that enable language-neutral object and class sharing across dynamic and static languages. The CoLoRS object model is a static-dynamic hybrid, which provides the efficiency of a static model with the flexibility of dynamic class modifications. To enable this, CoLoRS uses an extensible static model with versioning and type mapping.

In addition, CoLoRS implements a parallel, concurrent, and on-the-fly GC that is better suited for multi-VM memory management than extant GCs. CoLoRS GC is simpler than state-of-the-art on-the-fly GCs, does not require tight integration into a runtime, and imposes no system-wide pauses. Moreover, CoLoRS uses a synchronization mechanism that avoids the complexities of conventional approaches to monitor synchronization, while providing the same semantics and comparable performance. Both GC and synchronization in CoLoRS are designed specifically for cross-MRE sharing.

To investigate object sharing between dynamic and static OO languages, we integrate CoLoRS support within open-source, production-quality runtimes for Java and Python. We have evaluated CoLoRS efficacy using standard Java and Python benchmarks and found that CoLoRS extensions impose low execution time overhead. We also provide detailed experimental results for the CoLoRS GC algorithm and CoLoRS synchronization.

An important use case for CoLoRS is cross-language RPC. We have found empirically that CoLoRS can significantly (up to 2 orders of magnitude) improve the performance of such RPC systems as CORBA [50], REST [66], Thrift [143], and Protocol Buffers [129]. This is because using shared memory in the co-located case avoids expensive object serialization. The improvements in communication throughput and latency due to CoLoRS significantly increase end-to-end transaction performance in Cassandra [2] (a key-value database), and the Hadoop Distributed File System (HDFS) server [78].

In the sections that follow, we present the design and architecture of CoLoRS, describe the key components of our system (Section 7.2), including a language-neutral object/memory model, memory management, garbage collection, and synchronization support, as well as transparent object sharing via runtime/library virtualization. We then discuss CoLoRS empirical evaluation (Section 7.3), investigate how to implement CoLoRS support in an unmanaged language (C++) in Section 7.4, compare/contrast CoLoRS with related work (Section 7.5), and conclude (Section 7.6).

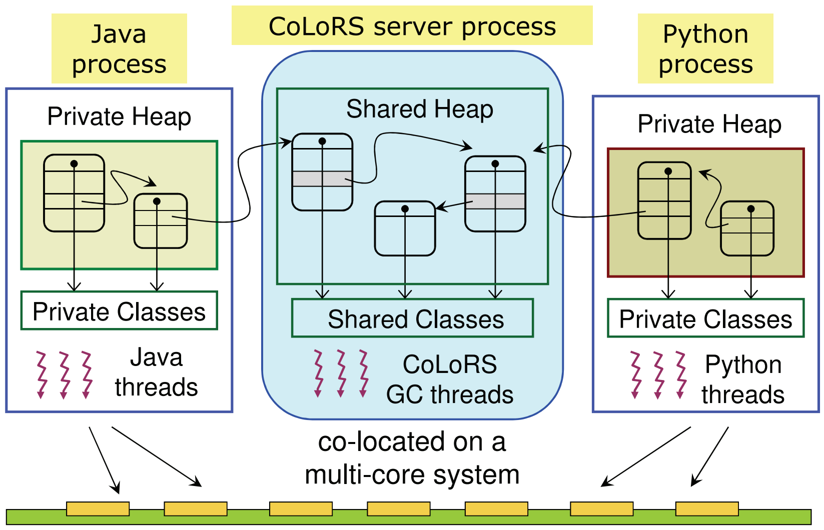


Figure 7.1: CoLoRS architecture. There is exactly one CoLoRS server process, which manages the shared memory segment and runs concurrent GC. Runtimes for different languages (Java and Python in this case) attach to the shared memory segment and allocate/use objects in the shared heap.

7.2 Design and Implementation

A primary design goal of CoLoRS is to provide type-safe, transparent, direct object sharing between co-located managed runtimes for different OO languages. This includes both statically-typed (e.g. Java) and dynamically-typed (e.g. Python) languages. The key challenge with providing such support are the major differences between language implementations, including object/class models, memory models, type systems, builtin types, standard libraries, and memory management (GC). For instance, dynamic languages support attribute (member) addition at runtime, while static languages permit class changes at compile-time only.

Figure 7.1 shows a high-level view of a CoLoRS system. In this example, two VM processes (one for Java and one for Python) are co-located on a multi-core system. There is exactly one CoLoRS server process which manages the shared heap (this includes the setup of the shared memory segment, data structure initialization, as well as support for garbage collection). Each VM process has its own private heap and a private object/class model and runs its applications threads. In the shared heap, there is a CoLoRS object/class model which is transparently translated to a private object/class model in each VM. All VMs map the shared memory segment at the same address in the virtual address space and use shared objects directly via pointers.

CoLoRS does not allow pointers from the shared heap to any private heap because of memory/type safety. In our experience, this restriction is rarely violated in standard libraries and most existing classes can be shared without any modifications.

Static (class) fields are not subject to sharing because they often represent local resources and sharing them would break resource isolation. For instance (object) fields, however, CoLoRS supports fully transparent sharing with regard to allocation, GC, field access, (virtual) method invocation, monitor synchronization, standard libraries, and class loading.

We do not support code sharing because that would require defining a VM-neutral language and checking whether two methods are equivalent, which in general is undecidable. Instead, CoLoRS guarantees type-safety for data/state sharing only. To reduce the programming effort associated with ensuring that the code/behavior matches across different languages, methods can be translated between languages automatically. Note that it is sometimes desirable to have different class implementations/interfaces in different VMs: standard libraries differ across languages and we do not want to unify them because programmers are used to existing libraries and there is a lot of legacy code written to them. Sharing only instance fields makes CoLoRS more practical as the code and static data do not have to match across languages.

A general approach we take in CoLoRS is to define a language-neutral, shared object model (with respect to non-static data) and then dynamically map it to each runtime-specific object model. To implement this, we virtualize all runtime components that rely on a specific object model. Modifications to runtimes are necessary to make object sharing transparent. In particular, CoLoRS needs to intercept all field accesses to handle shared objects correctly.

7.2.1 CoLoRS Usage

CoLoRS provides a simple application programming interface (API) for developers. The CoLoRS API for Java comprises the following methods in the $SharedMemory$ class (Python has equivalent API):

Object copyToSharedMemory(Object root);

Object allocate(Class objectClass);

```
Object allocate(Class containerClass, int length);

boolean isObjectShared(Object object);

ObjectRepository findOrCreateRepository(String key);

ObjectChannel findOrCreateChannel(String key);

Type getSharedType(Object object);
```

CoLoRS supports two ways of creating shared objects: via direct object allocation (the $allocate$ method) and via deep copying of a private object graph to shared memory (the $copyToSharedMemory$ method). The $allocate$ method has two variants: one for allocation of fixed-size objects and one for allocation of container objects (which takes the initial size of a container as a parameter).

Note that we do not support a state model where a thread can switch to the shared mode and issue regular object allocations to allocate in shared memory (as is done in related work on cross-JVM sharing [160]). The reason is that the state model requires complex rules specifying which allocations should target shared memory. For instance, in a JVM, we must exclude class loading, static initializers, and exception handling from leaking objects into shared memory.

CoLoRS provides two mechanisms to initiate communication between two runtimes: channels and repositories, both of which are named entities enabling exchange of a reference to a shared object. The $ObjectRepository$ class provides nonblocking get/set functionality while the $ObjectChannel$ class supports blocking send/receive cross-VM semantics. The following code fragments show an idiomatic repository usage for two Java processes. The client process:

```
ObjectRepository r = SharedMemory.findOrCreateRepository(“db”);
synchronized(r) { while(r.get() == null) r.wait(); }
```

The server process:

```
ObjectRepository r = SharedMemory.findOrCreateRepository(“db”);
synchronized(r) { r.set(root); r.notifyAll(); }
```

For object channels, we have a similar pattern but synchronization/waiting is not necessary because of the blocking behavior of send and receive.

Each repository holds a reference to its root object. Each channel has a fixed capacity for messages and blocks the sender when full. As long as a shared object is reachable from any repository, channel, or any VM, it stays alive. Unreachable shared objects are garbage collected. Channels and repositories are identified by a key (string).

The CoLoRS API enables reflective inspection of the shared type of a shared object via the $getSharedType$ method. We need this API method because in CoLoRS, expressions that evaluate to an object class, e.g. object.getClass() in Java, retrieve a private class to which a specific shared class currently maps. To see the shared class before mapping to a private class occurs, $getSharedType$ is used. Shared classes are regular objects – CoLoRS uses a three-level circular meta-data hierarchy that is fully traversable by programs wishing to inspect it.

A programmer can check whether an object is in shared memory via the $isObjectShared$ method. The system throws a $SharedMemoryException$ to prevent shared-to-private pointers as well as to signal type mapping failures, out-of-memory errors, and locking issues.

7.2.2 Shared Memory Segment

CoLoRS uses a dedicated process (CoLoRS server) to manage shared memory. There is one CoLoRS server per OS instance. This server creates, initializes, and destroys the shared memory segment, as well as runs concurrent, parallel GC. That is, GC continues to function even when no runtimes are currently attached. CoLoRS was designed to be scalable (GC, repositories) therefore having one server per host is not a limitation.

To use shared memory, runtimes attach to the shared memory segment (by mapping it to their virtual address space at the pre-defined, fixed address). The shared memory segment contains three spaces: metadata space (for state variables and synchronization), classes space (for shared types, repositories, and channels), and objects space (for garbage-collected shared objects). Each VM runs a separate CoLoRS thread which is responsible for collaboration with the CoLoRS server during GC.

CoLoRS intercepts all field accesses in the VMs and handles shared and private data differently. Private fields are read/written in a VM-specific way while shared fields use CoLoRS accessors.

7.2.3 The CoLoRS Object Model

CoLoRS employs an object model (OM) that aims at transparent and efficient cross-language object sharing, while supporting both static and dynamic languages. Our primary goal is maintaining the language-specific OM and object/class semantics while a VM interacts with shared objects. The rationale behind this is to avoid introducing a new unfamiliar programming model. In addition, CoLoRS combines certain characteristics of static and dynamic OMs in order to support the flexibility of a dynamic model while providing the efficiency and simplicity of a static model.

CoLoRS Type System

CoLoRS preserves language-specific type-safety without defining new typing rules by mapping shared types to private types. When mapping a shared type S to a private type P_1 in one VM and to private type P_2 in another VM, we guarantee that any field access permitted by P_1 does not violate the field typing constraints imposed by P_2 (and vice versa).

In the CoLoRS type system, every value is an object (there are no primitive types like in Java or C#). This is motivated by dynamic languages like Python and Ruby which treat everything as an object and therefore require that each value have a unique identity (address).

Unlike extant systems for cross-language data sharing, CoLoRS does not specify its own data definition language (DDL). Conventional approaches have resulted in a number of domain-specific DDLs, e.g. SQL in relational databases, WSDL in web services, and IDL in CORBA. The primary limitation of DDLs is their static nature and the necessity for a programmer to master another language. Instead, CoLoRS generates the shared data model automatically from the native language data model defined by the programmer. Moreover, this happens dynamically at runtime and only for types that are used in shared memory.

The CoLoRS OM strives to strike a balance between supporting diverse languages (both static and dynamic) and staying sufficiently close to each individual language so that costly runtime data conversions are avoided if possible. Another key design tradeoff is to support the flexibility of dynamic languages while leveraging the benefits provided by static typing. In fully static OMs (e.g. Java), object layout is completely described by classes, fields are efficiently accessed via offsets, each object consumes only as much memory as necessary for its attribute values, and the data model is fully documented by classes. On the other hand, in fully dynamic OMs (e.g. Python) classes do not describe object attributes, each object maintains a dictionary mapping attribute names to values, field access is expensive as it takes place via names, and space usage is suboptimal due to the redundancy across attribute dictionaries. However, unlike static OMs, dynamic OMs support dynamic attribute addition/removal as well as per-object attributes.

Several hybrid models have been introduced to mitigate the static-dynamic tradeoffs. A partially static/dynamic OM is used by Google AppEngine, where each object has a static part (fields described by a class) and a dynamic part (per-object dictionary). On attribute access, the system first tries to use a static field then falls back to an object dictionary on failure. Dynamically created attributes do not become part of the static model. A similar concept has been introduced to Python (via the __slots__ declaration). The JavaScript V8 runtime implements hidden classes to enable fast, offset-based attribute lookup while supporting dynamic attribute addition and deletion.

Hybrid OM and Versioning

CoLoRS OM is a static-dynamic hybrid, which can be described as an extensible static model with versioning and type mapping. Our goal is to keep CoLoRS OM as static as possible but still allow the flexibility of modifications (add/remove/change name/type of a field).

Shared classes are always created based on private classes when a private object gets allocated in (or copied to) shared memory. On each allocation in shared memory, we inspect the fields of the allocated object and look for a shared class being an exact match for a given type name and field set. If we do not find an exact match, we create a new class (or if a class with this name already exists, we create a new shared class version, having the same class name but a different field set). For example, suppose that we have the following class in Java:

```
class Employee { String name; double salary; }
```

and we perform shared allocation using:

```
Employee e = (Employee)SharedMemory.allocate(Employee.class);
```

If no $Employee$ class is present in shared memory yet, we create one, with two fields that correspond to the private $Employee$ class. Now assume that we add a new field to the $Employee$ class, say $Employee\ manager$; and we repeat the shared allocation as shown above. This time, CoLoRS will create a new version of the shared $Employee$ class, with three fields. Note that at any point in time there is exactly one private $Employee$ class (which may evolve in time) and there may be multiple versions of shared $Employee$ class (reflecting the schema evolution). Field removal is handled in a similar way.

Shared objects use shared classes to describe their layout. Different versions of a single shared class may have different layouts in memory and field sets. Shared classes are read-only, they do not change. However, shared objects may change their class pointers (from one version of a particular class to another version of the class). This can happen both in static and dynamic languages. For example, the following code in Python, which uses our two-field $Employee$ class:

```
e = shared_memory.copy_to(Employee('Smith', 100))
e.state = 'NY'
```

adds a new field (called $state$) dynamically. To support this in shared memory, CoLoRS creates a new version of the $Employee$ class and changes the current class of the e object to the new class version. Dynamic field removal (via del in Python) is handled similarly.

The advantage of versioning over a pure OO model is lower space consumption. In conventional OO systems, class evolution takes place via subclassing: to add or hide a field a new class is created that inherits from the old class. As a result, it is not possible to remove any attribute and space is consumed forever by unused fields. In contrast, with versioning, even if classes evolve, the newly-created objects always consume the optimal amount of space.

Type Mapping

To correctly handle multiple class versions in shared memory, CoLoRS uses type mapping. Each private class P in a VM always has exactly one version which, at any given moment, may be mapped to several different versions of class P in shared memory (a one-to-many relationship). Except for builtins (e.g. Integer, String), mapping only occurs between classes with the same name – programs in different languages must agree on package/module and class names. We map a shared field to a private field if and only if both have the same name and the same (or convertible) type. In dynamic languages, we map solely on the field name basis as there are no static types available.

Since type mapping is a relatively expensive process, we perform it lazily, once per shared-class-version, and maintain the mapping in a private hash table in each VM. We also use a reverse mapping table, to avoid shared-type lookup/matching on every allocation in shared memory. Note that on allocation, we need to obtain the shared type based on a private type. In contrast, when accessing a field in a shared object, we perform the mapping from a shared type to the private type.

When CoLoRS allocates a new object in shared memory, it tries to find a shared class version that exactly matches the private field set of the newly-allocated object. If no exact match is found, it creates a new shared class version. Consequently, newly-created objects do not contain fields that were removed from a private class due to its evolution. The rationale behind this is that we want to keep the object size in shared memory optimal. However, when mapping a shared class to a private class in a context other than allocation, we allow both private and shared fields to remain unmapped (if they do not have a match). When a VM uses an unmapped field in a shared object, we dynamically add a field to a class. To do so, we create a new shared class version that contains the previously unmapped field, and change the shared object's class pointer to point to the new class version. Note that the shared object's type does not change, as seen from the VM's perspective – all versions of a shared class always map to the same private class (with the same name).

Although CoLoRS supports dynamic changes, once the data model is stable, both space usage and field access work exactly like a fully static model. Also, in the CoLoRS OM, all object attributes are always present in its class and can be introspected via reflection.

Some VMs, such as Java, support class loading that makes it possible to have multiple classes with the same fully-qualified name. CoLoRS supports this via type mapping. One shared class can map to multiple private classes (e.g. we can map a single shared class named a.b.C to all private classes named a.b.C loaded by different class loaders).

Figure 7.2 shows an example where private class A evolves from a single-field class containing "int a" into a class with two fields, "int a" and "float b". Private class A has exactly one version (the newest one with both fields). Shared class A has two versions. Both shared versions are mapped to the private class A so that they can be uniformly used, despite being distinct types in shared memory. The shared objects space contains two objects of class A – one allocated for the old version of A and one allocated for the

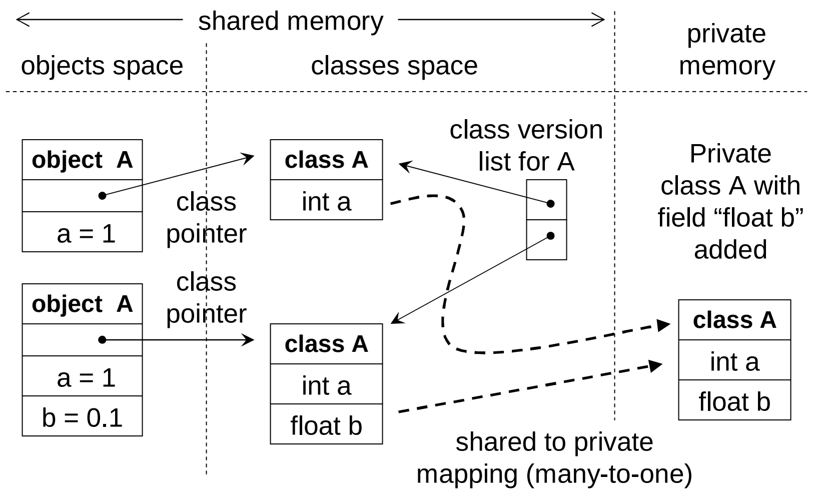


Figure 7.2: An example illustrating CoLoRS versioning and type mapping as private class A evolves by having a field added.

new version of A. Note that each shared object uses only as much space as necessary for its attribute set. Both objects have the same type in a VM, and the VM may access both fields (a and b) in both objects. On access to a non-existent field (b in this case) in older shared objects, CoLoRS will expand the object to make room for the new field (initializing the new field to 0).

Reconsidering the example in Figure 7.2 in the case when class A evolves by having the b field removed, we have a similar situation. Private class A again has exactly one version (the newest one, with one field a). Shared class A has two versions, both mapped to the same private type A. Field b remains unmapped as it can never be used by the VM and this field is simply ignored in those shared objects that have it. Note that newly-allocated shared objects do not reserve a slot for field b, thus using optimal amount of space. In contrast, OO inheritance does not allow removal of a field from an object (unused inherited fields continue to consume slots in objects). Field renaming is equivalent to field removal followed by a field addition.

Note that using CoLoRS cannot lead to broken program invariants because matching fields can never remain unmapped. Thus, if class implementations across languages match and preserve some invariant in each language, CoLoRS will preserve this invariant too.

Built-In Types and Libraries

CoLoRS provides full transparency for builtin types (e.g. strings, integers, lists, and sets). Builtin types differ significantly across languages and at the same time are frequently used by programs and libraries. CoLoRS preserves language-specific interfaces for builtin types by virtualizing the builtin implementation and/or standard libraries in each runtime. Library virtualization amounts to modifying the code of library methods so that these methods check whether any of the method arguments (including the receiver, if any) is a shared object and, if so, to execute a different implementation of the method.

CoLoRS defines a set of builtin types which we identify in Table 7.1 with their mappings in Java and Python.

We support 64-bit integers, which can be mapped to Python int and to any integer type in Java, both primitive, e.g. int, $short$, and reference, e.g. $Long$, $Integer$. Having only one integer type allows us to avoid complex rules for field mapping during schema evolution. For example, if we supported int and $short$ as distinct integer types in shared memory, then we would have to define complex semantics for changing the field type from int to $short$ and vice versa, i.e. when we create a new field dynamically and when we reuse existing integer field.

We use a similar approach in case of floating-point types, supporting only 64-bit IEEE floats. The CoLoRS 64-bit float can be used in Java as any floating point type, e.g. $double$ or $Float$. We do overflow/underflow checks when reading/writing integer/float fields requires conversion.

For non-container types, we also provide $boolean$ and $string$. As in Thrift [143], CoLoRS defines three container types: $list$, set, and map. Containers are untyped (i.e. may contain objects of different types at the same time). This is because we cannot automatically infer the container element type (at least in Java and Python), even if the container is not empty. To support a compact byte array representation we provide the $binary$ type, suitable for blobs. Note that in Java, a shared $list$ can be used as an array (of any type) and as a $List$. The rationale behind this is transparency – we want to support Java arrays even though CoLoRS and Python do not have arrays so that we do not change the Java programming model. Non-container types (integer, float, boolean, and string) are immutable. Builtin objects always have exactly one version, exactly one mapping to a private type, and do not have any programmer-visible fields.

In order to use shared objects along with private objects in a single hash-based container, hash codes and equal-to methods must agree across runtimes. We unify them for Java and Python builtin types. For shared objects, CoLoRS provides default hash code generation, equal-to methods, and less-then methods (all based on object addresses). They can be overridden by programmers.

For programmer convenience, CoLoRS automatically copies non-container types (e.g. integer, string) to shared memory. On field assignment/array store, the system checks whether the assignment uses a private r-value and a shared l-value. If so, and the r-value is of a non-container type, CoLoRS silently calls the $copyToSharedMemory$ method on the r-value, instead of throwing an exception. This mechanism is particularly useful for constructors.

Static Languages

In static languages, object fields are typed and typically accessed using field offsets. Since CoLoRS uses a mostly-static OM, it also identifies fields in shared objects by their offsets. Private and shared field offsets may differ so it is necessary to map between them. Unidirectional mapping from the private offset to the shared offset is sufficient because VMs always access shared fields using the context of a private type.

Shared	Java	Python
integer	byte, short, int, long, char, Byte, Short, Integer, Long, Character	int
float	float, double, Float, Double	float
boolean	boolean, Boolean	bool
string	String	str
binary	byte[]	bytearray
list	List, ArrayList, Object[], int[], float[], T[], ...	list, tuple
set	Set, HashSet	set, frozenset
map	Map, HashMap	dict

Table 7.1: Builtin types supported by CoLoRS and their mappings to Java and Python builtin types. For transparent and convenient use by programmers, multiple mappings are possible per shared type.

To make this mapping efficient, we associate a field-offset-table with each pair (S,P) where S is a shared type mapped to private type P. Whenever we access a shared field in a shared object, we index the appropriate field-offset-table with the private field offset and obtain the shared field offset.

When inspecting a class of a shared object (e.g. via object.getClass() in Java) we always get a unique private class as a result. For example, $integer$ maps to $SharedInteger$ while $list$ maps to $SharedList$. However, to ensure transparency, shared builtins can map to multiple different private types. In OO languages, this can be implemented via multiple inheritance. For instance, if we can make $SharedList$ inherit from $List$, $Object[]$, $ArrayList$, etc. then representing shared $list$ as private $SharedList$ is correct in all possible mappings. However, some languages (e.g. Java) do not support multiple inheritance or inheritance of array types. We instead simulate both by modifying the runtime so that $SharedList$ can be cast to any of the private types that shared $list$ maps to. We apply a similar approach for $integer$ and $float$.

Each private class maps to a unique shared class. A general rule that we use is that whenever we allocate private type P as shared type S, we must later be able to use the shared type S as P.

Type mapping may cause class loading in a VM. This is because whenever we encounter an instance of a shared type T, which maps to a private type U, we must load class U. Thus, CoLoRS introduces a new class loading barrier (in VMs that use dynamic loading).

Since in static languages, the static type of a field is available, we permit certain conversions while mapping shared fields to private fields. Let us denote any private class to which a shared class S maps as map(S). For a given field of shared type S and of private type P, CoLoRS allows both upcasts and downcasts during mapping.

Upcasts occur if class P is a superclass of class map(S) or class map(S) implements interface P. For instance, we have an upcast when we map a field of shared type $list$ to a field of private type $List$ (because map($list$) = $SharedList$ and $SharedList$ implements the $List$ interface). Or we have an upcast when we map a field of shared type $string$ to a field of private type $Object$, because $Object$ is a superclass of class map($string$) = $String$. Upcasts are most useful to support interface-type private fields, such as $List$ in Java.

Downcasts take place if class P subclasses map(S). For example, there is a downcast if a field of shared type $list$ is mapped to a field of private type $String[]$, because $String[]$ subclasses $Object[]$ = map($list$). Thanks to downcasts, private arrays (whose elements are typed) can conveniently access shared lists (whose elements are untyped).

To ensure type safety, downcasts require a read barrier which checks the actual object type on each read access. Upcasts represent a covariant type operator (analogous to the array upcasts in Java) and therefore require a write barrier that checks the type of the stored object against the expected static type.

Dynamic Languages

In dynamic languages, fields are accessed by name (not by offsets) and static field types are not available. Therefore, when creating a new shared class or comparing to an existing one, CoLoRS relies on actual types of all non-null attributes in a particular object (i.e. the one being copied to shared memory). This results in type concretization – shared classes created by dynamic runtimes always have the most derived field types. Such concretized types can be later used by static runtimes without any problems because static runtimes allow upcasts during type mapping.

We ignore NULL fields as for them no static (concretized) type can be inferred. When looking for an exact type match (during copying to shared memory), we allow type conversions (upcasts and downcasts). No read barrier is necessary as dynamic languages do not guarantee any particular type for any field. However, each field store must verify the type of the stored object against an appropriate static shared type (via a write barrier).

When mapping a shared type S to a private type P, we do not map fields, as we do not have field types and offsets in P. Instead, we just create a hash table mapping field names to shared offsets. This speeds up attribute access (which is done via names). Since multiple private types can be mapped to a single shared type (e.g. $list$ and $tuple$ in Python both map to shared $list$), we employ multiple inheritance if possible (e.g. in Python) or we extend the runtime to simulate it for the types in question.

CoLoRS uses reverse mapping to avoid shared class lookup on each allocation. Reverse mapping can improve performance only if private instances of a single private class have similar attribute sets (a natural property but one that is not always enforced by dynamic languages). Otherwise, the system might end up relying on dynamic field addition frequently as some objects' types may be mapped to static types that have too few static attributes.

7.2.4 The CoLoRS Memory Model

CoLoRS defines a memory model (MM) that builds on and simplifies memory models supported by mainstream languages. CoLoRS MM is equivalent to the Java MM for programs that do not contain data races. Java programs that rely on $volatile$ and $final$

fields or other race-related aspects of the Java MM may work incorrectly with CoLoRS because shared object fields drop their Java-specific modifiers. Python does not define any MM so using CoLoRS cannot break extant Python programs.

Following the Java Memory Model (JMM) approach and recent standardization effort for the C++ MM [33], CoLoRS guarantees sequentially consistent semantics only to programs that are properly synchronized (i.e. those that do not contain data races). A data race occurs when multiple threads can access the same object field at the same time and at least one of them performs a write.

Similarly to Java and C#, CoLoRS provides monitor synchronization. Monitors provide mutual exclusion for threads and restrict re-ordering of memory accesses. Monitor entry has load acquire semantics (downward fence) while monitor exit has store release semantics (upward fence). Full memory fence is not supported in CoLoRS (following Java and C# design) – a pair of downward and upward fences does not constitute a full fence. In CoLoRS, monitors are fault-tolerant: if a VM dies while holding a monitor, subsequent acquisitions of this monitor do not result in a deadlock or access to corrupted data, but throw a runtime exception before entering a critical section.

Like the JMM (and unlike the C++ MM), CoLoRS must guarantee basic type- and memory-safety even in the presence of data races. Therefore, in CoLoRS, all pointer stores and loads are always safe (even with data races). This property is relatively easy to implement (an aligned machine-word-wide load/store is atomic on most architectures). This property is not strictly necessary for type-safety in case of primitive values, like integer or float, and therefore CoLoRS does not guarantee it for non-pointer fields. Operations like shared class creation or dynamic field addition are always thread-safe because they are rare and can be internally protected by a lock.

Note that CoLoRS MM avoids many of the complexities of the JMM by supporting only instance field sharing (no statics, no methods, no constructors) and ignoring field modifiers like $final$ and $volatile$. Unlike C++ MM, CoLoRS MM does not support atomic operations and the $trylock$ functionality, which simplifies the model significantly.

7.2.5 Monitor Synchronization

The CoLoRS synchronization mechanism is an adaptation and simplification of extant, commonly-used schemes, which are inadequate for CoLoRS because of their complexity, tight integration with VM services, and reliance on the ability to stop all the threads.

State-of-the-art high-performance VMs, like HotSpot JVM, use biased locking [135] to avoid atomic CAS operations in the common case. However, biased locking requires safepoint support – it occasionally needs to stop all the threads to recover from its speculative behavior. Safepoints are needed for bias revocation (when a thread must manipulate the stack of the current bias owner) as well as for bulk rebiasing (to walk all thread stacks to search for currently held monitors). One of the design goals in CoLoRS is to avoid stopping all VMs at once – such system-wide safepoints are inherently unscalable and introduce lengthy pauses. Therefore, biased locking is not suitable for CoLoRS.

Another commonly-used locking scheme is lightweight locking [135], which strives to avoid using OS primitives in the common case by relying on atomic CAS operations. We have investigated the efficacy of this approach and found that in modern OSes that provide futexes (fast user-mode locking primitives), lightweight locking performs worse that an OS mutex. In older OSes, OS-backed synchronization was slow because it required kernel entry/exit. Linux implements futexes that in the uncontended case perform one atomic CAS in user-mode for each pair of lock and unlock operations. In contrast, lightweight locking needs two atomic CASes [135] per uncontended lock-unlock pair. We have compared the performance of pthread-based locking and lightweight locking in the uncontended case. We measured the time needed to do one lock and one unlock. Our results show that lightweight locking is slower: on a dual-core Intel Core2 by 31%, and on a quad-core Intel Xeon by 45%. Therefore, we have designed CoLoRS to use OS primitives (POSIX mutexes based on futexes) directly.

Most extant monitor implementations (e.g. HotSpot JVM) reserve a word in the object header to assign a lock pointer to an object once a lock is needed. The presence of such a pointer leads to significant design complexity in extant systems because once the pointer is set, one can only clear it when all threads are stopped or the object has become unreachable. CoLoRS does not ever stop-the-world (halt/safepoint all threads in the system), hence we take a different approach.

Instead of using a pointer to a monitor, we hash the object address (shared objects do not move in CoLoRS) into a fixed-size table of monitors kept in shared memory. Since few objects are used as monitors at a time, it is unlikely that multiple simultaneously-locked objects will ever hash to the same monitor-table entry (i.e. hash conflicts are rare). To avoid deadlocks and decreasing concurrency level, we detect conflicts in the hash table and use a collision chain to ensure that each object gets a unique monitor. Hash-based locking is also used in GCJ [69] (GNU static Java compiler) in order to reduce the object header size. GCJ, however, uses both light- and heavy-weight locks.

We use mutex_trylock() to avoid blocking the acquiring thread in case there is a conflict in the lock-hash-table. We also tag lock-hash-table entries with an object pointer, once a lock is successfully acquired via mutex_trylock(). Each thread locking object O first checks if a hash-table entry is tagged with O. If so, the thread proceeds to mutex_trylock(). Otherwise, if the entry is tagged with P different than O, we re-hash to find another entry. If there is no tag there, we proceed to mutex_trylock(). Dead-object tags are cleared asynchronously by GC – for each conflict chain, GC creates and locks a new untagged chain entry, thus temporarily stopping chain expansion (all threads will block on mutex_trylock() in that GC-created entry). GC then clears the dead tags in the chain, and finally, notifies the blocked threads to repeat their locking from scratch (the re-do flag is set on the GC-created entry and the GC releases the mutex).

The above synchronization scheme can be transparently integrated into Java based on Java monitors. Python does not support the monitor abstraction (locks are not associated with objects) and therefore needs to be extended with dedicated API for monitors (similar to Java).

7.2.6 Garbage Collection

Since CoLoRS targets multi- and many-core systems and avoids system-wide safepoints, the most appropriate GC algorithm for shared objects is parallel (i.e. using multiple GC threads), concurrent (i.e. performing most work without stopping the application), and on-the-fly (i.e. stopping at most one thread at a time) GC. In addition, CoLoRS needs a non-moving, mark-sweep-style GC because some runtimes (e.g. Python) assume that objects do not move and other ones (e.g. Mono for C#) use conservative stack scanning.

We have found extant on-the-fly mark-sweep GCs to be unsuitable given the CoLoRS architecture and requirements. Therefore, we have designed a variation of snapshot-at-the-beginning (SATB) GC, which is parallel, concurrent, and on-the-fly.

The state-of-the-art in on-the-fly GC systems include those that employ the Doligez-Leroy-Gonthier [60] algorithm and its extensions by Domani et al. [61, 62] for generational heap layout and multiprocessors without sequential consistency.

State-of-the-art, snapshot-based, on-the-fly GC algorithms require multiple (three to start the collection cycle) system-wide handshakes with all the threads. The mutators must check whether they need to respond to handshakes regularly during their normal operation. For scalability, we designed CoLoRS to work at the granularity of VMs, not individual threads. The handshakes would require keeping track of all threads in all VMs. In addition, we do not want to require VMs to implement the per-thread handshake-polling mechanism, as it is not generally supported in VMs.

A design goal of CoLoRS GC is to abstract away private VM memory management to one operation: shared root report, without imposing any specific implementation details. As a result, we have designed an on-the-fly GC that does not use handshakes and works at the VM level (not thread level). In addition, the CoLoRS GC is simpler (as it does not have any phase transitions) and guarantees termination (some previous algorithms unreliably depend on the relative speed of the collector and mutation rate for termination).

CoLoRS uses thread-local allocation buffers (TLABs) to reduce allocation cost. Each thread performs bump-pointer unsynchronized allocation in its own TLAB. Once the TLAB is exhausted, it is retired, and the thread requests a new one. VMs request

TLAB- and large-object-allocation directly from the object space. The freelist contains all unallocated blocks whose size is at least the TLAB size. The freelist is protected by a lock.

GC Algorithm

Our GC comprises four concurrent phases: flag clearing, root report, marking, and sweeping. The CoLoRS server initiates a new GC cycle as soon as the heap usage crosses a specified threshold. The main GC thread is awoken by an allocating thread once this happens. CoLoRS GC imposes no pauses. If a VM is capable of reporting shared roots without causing internal pauses (e.g. as Python can), then the system never needs to pause any threads.

Flag clearing. The main GC thread first clears all GC-related flags in the heap. This operation is fully concurrent. Each object has three GC flags: pending (i.e. it needs to be recursively marked), marked (i.e. it has been recursively marked), and recent (it has been recently allocated).

Unlike in extant SATB GCs, in CoLoRS, the snapshot mode is active all the time. This simplifies the algorithm as it avoids complex state transitions and handshakes. The snapshot mode means that all objects are allocated live (i.e. with the recent flag set) and mutators use a write barrier: on pointer stores they mark the overwritten pointer as live (i.e. they set the pending flag). When GC scans a live object it sets its marked flag. During the flag-clearing heap scan, the main GC thread also computes a fully-balanced heap partitioning that is used later on for parallel scanning. The key system invariant is that it is always possible to sequentially scan all blocks in the heap, without any synchronization. We carefully design allocation procedures so that we do not break this invariant.

GC flag clearing has a similar effect to activating the snapshot mode from scratch in other algorithms, but does not require handshakes. Once GC flags are cleared, the main GC thread requests root dumps from all attached VMs.

Root report. Each VM must be able to identify pointers into shared memory in its private heap/stacks in an efficient way. In VMs using tracing GC this is straightforward – we either scan the whole heap (non-generational GC) or use a card table (generational GC). In the latter case (e.g. in Java), we extend the card table so that we can quickly find not only pointers from the old generation(s) to the young generation but also pointers from the old generation(s) to shared memory. To report shared roots in this case, we simply trigger a fast minor collection and efficiently find all pointers to shared memory.

In VMs which use reference counting GC (e.g. cPython), CoLoRS can track shared roots as they are created and destroyed, thus being able to report them any time without any processing. For each shared reference, we create a small proxy object in private memory with reference count set to one. Once the proxy object becomes unreachable (which we know immediately thanks to reference counting) we reclaim it and forget the shared root. Note that only private references can exist to the proxy object since there are no shared-to-private pointers.

CoLoRS requests roots from each VM and waits until all reports arrive. To report a shared root, a VM sets the object's pending flag. To ensure store visibility, a memory fence takes place on both sides once the reporting completes. CoLoRS does not use timeouts because it detects VM termination in a reactive way via TCP/IP sockets. Termination is noticed right away and the exited VM is removed from the waiting-for-roots list.

Marking. As soon as all roots are reported, the main GC thread initiates parallel, concurrent marking done by several worker GC threads. Each worker thread scans its own heap partition looking for pending objects, and recursively marks them using depth-first search. To ensure dynamic load balancing during marking, worker GC threads employ randomized work stealing. GC threads use barrier synchronization to meet at subsequent GC phases.

Once first marking completes, the main GC thread enters a loop. During each iteration, CoLoRS performs parallel, concurrent marking from pending objects. However, this time it stops marking the object graph once its sees an object with the recent flag set. The loop terminates when no new objects have been marked. Stopping marking on recently-allocated objects guarantees GC termination – there is a finite number of “old” objects in the heap when the GC starts, and all the newly-allocated objects are being flagged as recent. Therefore, GC must finish in a finite number of steps.

This scheme is correct because after the 1st iteration, a recently-allocated object cannot have a pointer to an object that is live but otherwise unreachable and invisible to GC (and thus it cannot be incorrectly left unmarked). Note that such a situation may occur during the first marking pass, which marks from the VM roots. Our snapshot write barrier (SATB WB) does not capture root pointer updates – it only captures heap stores. Suppose that root r points to object O, and a new object N is allocated having its only pointer set to O. If root r is later updated to point to N, we end up with a newly-allocated object N that has a pointer to a live object O that is reachable only through N. The reason for this is that we do not notice root updates. Such a situation is impossible from the second marking on, as during 2nd and subsequent markings we ignore roots and mark from the pending flags only (i.e. from heap objects that are protected by SATB WB). Reconsidering our example in the heap context: object O is marked as pending on r update, and will be marked/scanned even if we stop marking on object N (which has its recent flag set).

Sweeping. As soon as the marking loop terminates, CoLoRS moves on to concurrent, parallel sweep. Each worker GC thread scans its heap chunk trying to find the first potentially-free (candidate) block. This scan is done without synchronizing with mutators that are actively allocating objects. Once a GC thread finds a candidate block, it acquires the freelist lock and continues the scan as long as it encounters reclaimable blocks. Finally, it removes all found dead blocks from the freelist and inserts one coalesced block into the freelist. The GC thread releases the freelist lock and looks for the next candidate block. Our GC-mutator contract guarantees that all block headers are always parsable.

7.2.7 Implementation Details

CoLoRS can work under any OS that supports adequate IPC functionality. We have implemented CoLoRS in HotSpot JVM 1.6 and cPython 3.1 under Linux.

The first step in the process of extending a VM with CoLoRS support is to determine the VM object/class model, its relationship to the CoLoRS OM, memory management (GC) algorithm(s), and operations that use objects, typically field access, method calls, synchronization, etc. Next, we define type mapping for builtins and user-defined types, and add any runtime extensions (such as multiple inheritance) to support it. The next step is heap access virtualization which amounts to extending an interpreter, a JIT compiler, or both, to provide a separate control path for handling shared objects. Depending on a VM, other components may need similar extensions, e.g. the GC subsystem. Typically, we must intercept all program instructions that read/write heap objects. Next, we insert calls to the CoLoRS API along the newly added control paths. This step translates VM-specific operations into VM-neutral operations (e.g. getting an attribute by name into getting a field by offset). Lastly, we add GC runtime support – we implement a dedicated CoLoRS thread and the shared-root-dump operation in the private GC system.

7.2.8 Shared Memory Layout

The CoLoRS shared memory segment contains three spaces: metadata, classes, and objects. The objects space is a garbage-collected mark-sweep heap with TLAB/freelist allocation. The classes space is a bump-pointer space for immortal objects that contains shared classes, class version lists, and registered object repositories/channels. The metadata space contains pointers to all builtin types (in the classes space), pointers to the repositories/channels hash tables (mapping names to repositories/channels), a pointer to class versions hash table (mapping names to class version lists), as well as user-level monitors, internal system locks, the freelist head, space usage statistics, and the bump-pointer top (for the classes space).

Each CoLoRS monitor has its POSIX mutex and condition variable. We use the PTHREAD_PROCESS_SHARED flag to make the POSIX mutexes and conditions work across OS processes. In addition, monitors use the recursion count (to avoid re-locking by the same thread) as well as owner ID (VM ID plus thread ID).

The CoLoRS server maintains additional state (metadata) in private memory to manage GC threads, and to track the attached VMs. For each attached VM, there is a dedicated monitoring thread, which detects VM termination using an open TCP/IP connection to a VM. On VM termination, the monitoring thread receives an error when reading from a closed socket. Note that OS-level IPC (e.g. sockets) is the only reliable way of detecting process termination without resorting to timeout/keep-alive solutions. This is because in Unix systems certain signals (e.g. the KILL signal) cannot be intercepted.

We group class versions into lists based on their name. Object repositories/channels and classes are permanent entities – we do not collect them as they are small and reusable. Object repositories/channels are treated as GC roots during GC.

GC flags are implemented as one-byte-wide fields because of concurrent access. We assume that writes issued by a particular thread are visible to other threads in the order they are issued (sequential consistency guarantees this).

The objects space is a contiguous sequence of blocks. Each block can be an object, a free chunk (part of the freelist), or a TLAB. The block header contains two fields: block length and block type. This enables quick traversal of the heap without parsing actual objects – a key property for our concurrent GC. TLAB blocks contain an owner ID, which identifies the VM that is currently using the TLAB. This enables us to reclaim TLABs orphaned by asynchronously terminated VMs.

To provide transparent object sharing, CoLoRS intercepts all VM operations that access heap memory. To efficiently check whether an object is shared, CoLoRS uses a constant border between private and shared area in the virtual memory. Each memory-related operation, such as field access, compares the pointer value against this constant border.

7.2.9 HotSpot JVM

In static runtimes with high-performance, adaptively optimizing compilers, border-checks may be expensive as they make the intermediate code larger and more difficult to optimize. Therefore, in our CoLoRS implementation in the HotSpot JVM server compiler, we compile methods in two modes: CoLoRS-aware and CoLoRS-safe. The CoLoRS-aware mode is used for methods in which shared memory has been determined (via profiling during interpretation) to be commonly-used. For such methods, border-checking overhead and the additional code that handles the shared pointers are acceptable.

The remaining methods (a vast majority in practice) are compiled in the CoLoRS-safe mode, where private pointers are the common case. The CoLoRS-safe methods contain only the minimum number of border-checks needed to take a trap on shared pointers. Such traps deoptimize the method and recompile it as CoLoRS-aware, running the method in the interpreted mode in the meantime. The CoLoRS-aware methods use fast upcalls to C to handle shared pointers (CoLoRS is implemented in C). If fast upcalls fail (e.g. because class loading is needed), we bail out to the interpreter.

In CoLoRS-safe methods, we combine null checks with shared-border checks. Assuming that shared memory area is at lower virtual addresses than the private area, checking if a pointer is below the border detects both NULL pointers and shared pointers. If the check passes, we trap to the interpreter, which finds the actual cause of a trap itself (the trap cost is not a problem as it is the uncommon case path). In CoLoRS-aware methods we guard virtual method calls to prevent calling into a CoLoRS-safe method with a shared receiver (such calls need a trap). CoLoRS-safe methods must translate user-provided null checks into null-and-border checks to avoid eliding border checks along with null checks.

We also perform approximate data flow analysis which conservatively computes all methods which can operate on a pointer to a shared object. The analysis exploits the fact that shared pointers can only be produced by the methods from the CoLoRS API. We dynamically and incrementally build the call graph as classes are loaded. In the graph, nodes represent methods and there is an edge from node m to n, if method m can pass/return a reference to method n. In case of interface methods, we have additional edges leading to all implementors of a particular method. We divide all loaded methods into two classes: private and potentially-shared. Private methods can never reach shared objects. If any potentially-shared method contains the $putstatic$ bytecode, then we assume all methods containing the $getstatic$ bytecode to be potentially-shared. Otherwise, if a method is reachable from a potentially-shared method in the call graph, that method is also considered potentially-shared. Potentially-shared methods are compiled as either CoLoRS-aware or CoLoRS-safe, depending on the profiling data. Private methods do not contain any instrumentation. If class loading makes a previously-private method potentially-shared, we make the method non-entrant and recompile it.

CoLoRS intercepts all bytecode instructions that access objects in the heap (both fields and object header): putfield, getfield, arrayload, arraystore, invoke, monitor-related ones, arraylength, and objectclass. We extend the HotSpot template interpreter and the server compiler (both targeting amd64). In addition, we virtualize the HotSpot runtime written in C (biased locking, GC, class loading, JNI, JVM, JMM, JVMTI). Several internal classes are not allowed to be in instantiated in shared memory (e.g. Thread, ClassLoader) – they are VM-specific and do not make sense in the context of other VMs.

7.2.10 cPython Runtime

We virtualize shared objects via private proxy objects, each containing a forwarding pointer to a shared object and a normal Python header (comprising private type and a reference count). This design choice is dictated by the fact that Python uses reference counting GC and CoLoRS uses tracing GC (so there is no reference counts in shared object headers). The cost of one level of indirection is compensated by the fact that we do not need to perform type mapping on each shared object access – proxy objects have their private type computed once. All proxy objects have the same size and are bucket-allocated in a dedicated memory region (for fast border checks). Deallocation takes place once a reference count drops to zero. Thus, the number of proxies never exceeds the number of private-to-shared pointers. Finding shared roots in such a setting is fast and amounts to a linear scan of the proxy object region.

Proxy objects also simplify Python runtime virtualization, as the Python interpreter dispatches basic operations such as field access, method call, and operator evaluation, based on object type (note that proxies already have the proper private type set). We provide a new private type for each builtin shared type, and the interpreter automatically invokes the right implementation (shared/private). Python VM allocates only one global TLAB because the interpreter is single-threaded and simulates multi-threading by context-switching between program threads. The Python runtime component most complex to virtualize are standard libraries and builtin types, which provide rich, complex interfaces (e.g. for sorting, concatenation, set algebra, etc).

7.3 Experimental Evaluation

An important practical use case for CoLoRS is improving communication performance of RPC in the co-located case. We evaluate CoLoRS in this context because there are cross-language RPC frameworks, such as CORBA, Thrift, Protocol Buffers, and REST, to which we can compare. CoLoRS, however, provides significantly more functionality over extant cross-language RPC systems by enabling direct, type-safe, and transparent object sharing.

We compare CoLoRS-based RPC against extant RPC frameworks in terms of communication performance (i.e. latency and throughput). We also evaluate end-to-end server-client performance (response time and transaction rate) for two applications: Cassandra and HDFS. Finally, we measure the overhead of CoLoRS in programs that do not employ shared memory, using standard community benchmarks for Java and Python.

7.3.1 Methodology

Our experimental platform is a dedicated machine with a quad-core Intel Xeon and 8GB main memory. Each core is clocked at 2.66GHz and has 6MB cache. We run 64-bit Ubuntu Linux 8.04 (Hardy) with the 2.6.24 SMP kernel.

We use HotSpot JVM from OpenJDK 6 build 16 (April 2009) compiled with GCC 4.2.4 in the 64-bit mode. Our configuration employs the server (C2) compiler, biased locking, and parallel GC (copying in the young generation and compacting in the old generation). For the Python runtime we use the open-source cPython 3.1.1 (released August 2009) compiled with GCC 4.2.4 in the 64-bit mode.

To measure CoLoRS overhead in Java, we use DaCapo'08 and SPECjbb ('00 and '05). We set the heap size to 3.5x the live data size so that GC activity does not dominate performance and so that we capture all sources of overhead. We use the default input for DaCapo and 5 warehouses, with 90s runs, for SPECjbb.

In Python, we evaluate CoLoRS overhead using PyBench (a collection of tests that provides a standardized way to measure the performance of Python implementations), a set of Shootout cPython benchmarks (from [49]), and PyStone (a standard synthetic Python benchmark).

In all experiments, we repeat each measurement a minimum of seven times. For experiments that employ shared memory, we perform sufficient iterations to guarantee

	Thruput in calls/ms; CoLoRS/RPC in parenthesis			
RPC	boolean	integer	float	string
CORBA	173.22 (11)	82.67 (26)	83.20 (27)	75.96 (15)
ProtoBuf	31.73 (59)	30.98 (70)	34.32 (65)	26.43 (43)
REST	23.17 (81)	22.45 (97)	21.89 (102)	22.94 (50)
Thrift	237.04 (8)	283.23 (8)	274.37 (8)	149.08 (8)
CoLoRS	1876.08 (1)	2175.32 (1)	2231.45 (1)	1144.87 (1)

Table 7.2: Throughput for the microbenchmarks for builtins. For each data type, we show the throughput in calls per millisecond; in parentheses, we show the CoLoRS/RPC throughput ratio.

that GC is performed by CoLoRS. We report average values. The standard deviation is below 5% in all cases.

CoLoRS reserves 256MB in shared memory for objects and 64MB for classes. We use 32KB TLABs, and 2 parallel GC threads. In each experiment, we employ two co-located runtimes: Python and Java. Whenever running an unmodified (CoLoRS-unaware) JVM, we set its heap size to 300MB so that its private memory is comparable in size to the shared memory.

Note that our results underestimate CoLoRS potential since we implement CoLoRS in Python 3.1 and compare its communication performance with RPCs running on Python 2.6. This is because the RPC frameworks that we use have not yet been ported to Python 3.1. To quantify this difference we evaluate the performance of Python 3.1 relative to Python 2.6. The last column in Table 7.7 shows the overhead of Python 3.1 relative to Python 2.6 across our set of benchmarks. On average, Python 3.1 is slower by 20%.

	Thruput in calls/ms; CoLoRS/RPC in parenthesis			
RPC	tree:1	tree:2	tree:3	tree:4
CORBA	14.67 (13)	4.68 (15)	1.83 (17)	0.86 (17)
ProtoBuf	2.85 (68)	0.88 (78)	0.36 (85)	0.17 (91)
REST	8.73 (22)	2.66 (26)	0.91 (34)	0.31 (49)
Thrift	15.38 (13)	4.27 (16)	1.80 (17)	0.87 (17)
CoLoRS	193.66 (1)	68.61 (1)	30.61 (1)	15.08 (1)

Table 7.3: Throughput for the microbenchmarks for user-defined types. For each data type, we show the throughput in calls per millisecond; in parentheses, we show the CoLoRS/RPC throughput ratio. $tree : n$ means the type is a full binary tree of depth n.

	Latency in ms; RPC/CoLoRS in parenthesis			
RPC	boolean	integer	float	string
CORBA	0.62 (14)	0.65 (19)	0.62 (14)	0.63 (14)
ProtoBuf	0.22 (5)	0.31 (9)	0.21 (5)	0.23 (5)
REST	3.89 (90)	3.89 (113)	4.00 (89)	3.92 (90)
Thrift	0.09 (2)	0.10 (3)	0.11 (3)	0.12 (3)
CoLoRS	0.04 (1)	0.03 (1)	0.04 (1)	0.04 (1)

Table 7.4: Latency for the microbenchmarks for builtins. For each data type, we show the latency in milliseconds; in parentheses, we show the RPC/CoLoRS latency ratio.

	Latency in ms; RPC/CoLoRS in parenthesis			
RPC	tree:1	tree:2	tree:3	tree:4
CORBA	0.68 (17)	0.82 (15)	1.13 (17)	1.92 (19)
ProtoBuf	0.55 (14)	1.32 (23)	2.90 (44)	6.02 (58)
REST	4.07 (101)	4.80 (85)	7.35 (111)	9.94 (96)
Thrift	0.19 (5)	0.35 (6)	0.74 (11)	1.38 (13)
CoLoRS	0.04 (1)	0.06 (1)	0.07 (1)	0.10 (1)

Table 7.5: Latency for the microbenchmarks for user-defined types. For each data type, we show the latency in milliseconds; in parentheses, we show the RPC/CoLoRS latency ratio. $tree : n$ means the type is a full binary tree of depth n.

7.3.2 CoLoRS Impact on Communication Performance

We first evaluate the performance potential of CoLoRS-based RPC using communication microbenchmarks with a range of message types and sizes. We implement equivalent microbenchmarks using RPC frameworks for CORBA, Thrift, Protocol Buffers, and REST. We compare RPC latency and throughput (call rate).

For the implementation of the microbenchmarks, we use a Python client and a Java server. Whenever possible we employ RPC methods with fully symmetric input and output (i.e. returning a data structure similar to the data structure passed in as an argument). This ensures that the server and the client exercise data structure (de-)serialization in a symmetric way.

To evaluate RPC throughput, we vary method input/output size between 1 to 1024 units and measure mean time per method call. Next, we use least-squares linear regression to compute throughput from the coefficients in the equation $time = latency + size/throughput$. We calculate latency as the mean time needed per call for unit input/output. We employ this methodology because we have observed that for small input sizes the function $time(size)$ is sometimes non-linear and approximating it by a line leads to an inaccurate latency estimation.

Each RPC method call takes a list as input and returns a list as output. List sizes vary between 1 and 1024. For each list size we do 10 experiments and use their average in the calculation above. We use several different objects as list elements, including

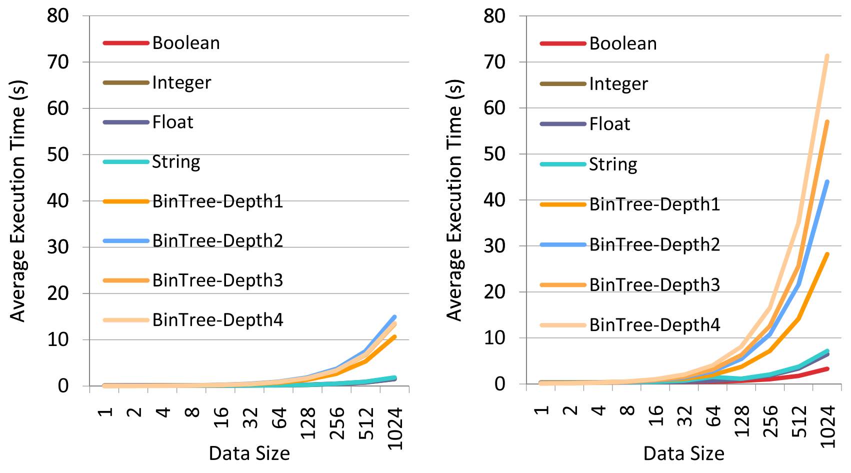


Figure 7.3: Average execution time (in seconds) for CoLoRS (left) and CORBA (right) experiments.

built-in primitive types (string, integer, float, and boolean) and user-defined types. For the latter we employ binary trees, the depth for which ranges between 1 and 4 levels, and each node contains 4 primitive fields. This enables us to investigate both shallow- and deeply-linked data structures. The above choice is also dictated by the limitations of extant RPC frameworks which support a small set of builtins and do not support recursive data structures. (Note that CoLoRS provides a richer and more flexible object model than these RPC systems.)

We implement an RPC endpoint in CoLoRS as a message queue on which a server waits for messages (call requests). Each message is an object encapsulating input and output. A client issues a call by allocating a message object (and the associated input) in shared memory, enqueuing it, and notifying the server. The server removes the request from the queue and generates the output in shared memory. Finally, the server notifies the client that the result is ready (as the output field in the message object).

For all experiments, we report throughput as the number of calls per millisecond, and latency in milliseconds. Due to space constraints, we only present timings graphs that compare CORBA to CoLoRS. This data is shown in Figure 7.3. The x-axis is message size and the y-axis is time in seconds. This data is representative of all of the RPC experiments. We summarize the latency and throughput of each below.

Table 7.2 and Table 7.3 show throughput across all microbenchmarks and RPC systems. We report both absolute values and relative improvement due to CoLoRS. Table 7.4 and Table 7.5 use a similar format but presents results for our latency measurements.

CORBA. The Common Object Request Broker Architecture (CORBA) [50] standardizes object-oriented RPC across different platforms, languages, and network protocols. A client and a server use automatically-generated stubs and skeletons to (de)marshall arguments and return values for methods specified in the Interface Definition Language (IDL). To implement our CORBA benchmarks, we use the $org.omg.CORBA$ package and the $idlj$ compiler in Java and the $Fnorb$ module and the $fnidl$ compiler in Python. Our measurements indicate that, compared to CORBA, CoLoRS achieves 11–27 times better throughput and 14–19 times lower latency.

Thrift. Thrift is a framework originally developed at Facebook for scalable cross-language RPC. Like CORBA, Thrift requires a language-neutral interface specification from which it generates client/server template code. However, Thrift is simpler and much more lightweight than CORBA. We use Apache Thrift version 2008/04/11. Our experiments show that CoLoRS improves throughput by 8–17 times and latency by 2–13 times, over Thrift. We also find that Thrift achieves much better performance for builtin types than for user-defined types.

Protocol Buffers. Protocol Buffers (PB) are a language-neutral, platform-neutral, extensible mechanism for serializing structured data, developed by Google engineers as a more efficient alternative to XML [129]. To use PB, developers specify message types in a $.proto$ file, and a PB compiler generates data access classes that allow to parse/encode objects into a bytes buffer/stream. We use PB version 2.2.0, which includes message parsers and builders but does not support RPC. Therefore, we implement RPC on top of PB by using PB serialization and communication over TCP/IP sockets. We maintain a single TCP connection throughout each experiment. Each message that we send from a client to a server, contains a method tag, message length, and PB-serialized data structure (method input). CoLoRS improves the throughput of PB-RPC by 43–91 times and latency by 5–58 times.

REST. REpresentational State Transfer (REST) [66] is a client-server architecture based on HTTP/1.0 where requests and responses are built around the transfer of representations of resources. REST provides stateful RPC by exchanging documents that capture the current or intended state of a resource. Individual resources are identified in requests by URIs. In our benchmarks, we define a single resource stored on a server and identified by `http://localhost:8080/db/items`. A representation of this resource is an XML document containing all stored items. Clients send GET requests to the resource URI, and parse the resulting XML document. This document contains a varying number of items (1–1024), where each item is either a primitive or a user-defined object. We employ the Python $restful\_lib$ to implement the client and the Java $restlet$ (version 1.1.6) for the server. Relative to REST, CoLoRS throughput is 22–102 times higher and latency is 85–113 times lower. REST has the highest latency among all of the RPC technologies that we investigate because of the verbose data format and parsing overhead of XML.

7.3.3 CoLoRS Garbage Collection

We gathered basic GC statistics for our Java-Python microbenchmarks. The results are similar across all the payloads that we use (described in the previous section). Below we discuss the experimental data obtained for 4-level binary trees.

We set the GC triggering threshold to 70%. Average time between subsequent GC cycles is 1458ms while average GC cycle time is 325ms (GC is active 18% of the time). Note that GC runs concurrently in a separate process. The clearing phase takes 94ms on average (29% GC cycle). The root dump phase was 1.2ms on average (below 0.4% GC cycle). In the HotSpot JVM, each root dump request causes a STW pause which averages at 0.8ms (with the maximum pause of 2.9ms). In cPython there is no pauses. The marking phase takes 116ms on average (36% GC cycle). Two object graph scanning iterations suffice on average (the maximum is 3). The sweep phase averages at 113ms (35% GC cycle). The dominating GC phases are marking, sweeping, and clearing, each taking around 1/3 of each GC cycle.

7.3.4 CoLoRS Impact on End-to-End Performance

To lend insight into the CoLoRS potential when used by actual applications, we investigate two popular server-side software systems: Cassandra [2] version 0.4.1 and HDFS [78] version 0.20.1. Cassandra is a highly scalable, eventually consistent, distributed, structured, peer-to-peer, key-value store developed by Facebook engineers. HDFS is the Hadoop Distributed File System – a file system server that provides replicated, reliable storage of files across cluster resources. Both of these systems are employed for a wide range of web applications, e.g. MapReduce, HBase (open-source BigTable implementation), email search, etc.

Cassandra and HDFS both expose Thrift-based interfaces. These interfaces provide a set of query/update methods which use relatively complex data structures (e.g. maps). Query methods are natural candidates for in-memory result caching, recently a common approach to scaling up servers (e.g. MemchacheD, MySQL cache). If caching is used, then in the common case (i.e. on cache hit), server processing is minimal and therefore communication constitutes a large portion of the end-to-end performance.

In systems with in-memory caching, CoLoRS can improve performance in two ways. First, it can reduce RPC cost by avoiding serialization. Second, part of the in-memory cache can be kept in shared memory – immutable objects such as strings can be shared by multiple clients without the risk of interference. As a result, CoLoRS can provide copy semantics without actually copying data. To investigate both these scenarios, we extend Cassandra and HDFS with in-memory caches for particular queries and evaluate the efficacy of using CoLoRS for these queries, on end-to-end performance. Note that when caching is used, the benchmarks exercise not only copying to shared memory but also frequent access to shared objects (which includes translation overhead).

For Cassandra, we implement caching for the get_key_range query (parameterized by table name, column family, start value, end value, maximum keys count, and consistency level). The query returns a list of keys matching the given criteria. Updaters, such as insert and remove, detect conflicting modifications and invalidate the cache ac-

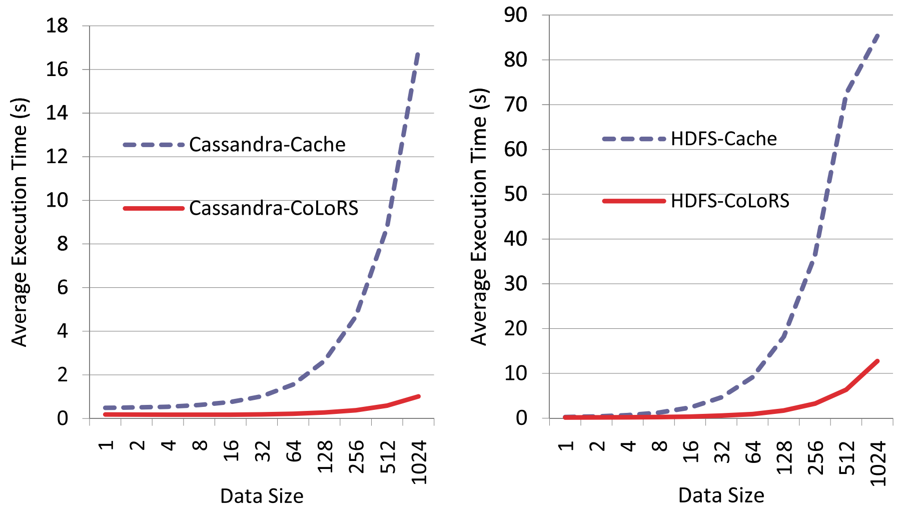


Figure 7.4: Average execution time (in seconds) for Cassandra (left) and HDFS (right) vs. CoLoRS.

cordingly. The cache is kept on the server and maps inputs (serialized to a string) to responses. Cached responses are partially in shared memory (strings are immutable). Thus, CoLoRS has the potential for improving performance by avoiding serialization and reducing copying overhead.

For HDFS, we implement an in-memory cache for the $listStatus$ call, which, given a directory name, generates a list of $FileStatus$ objects, each describing file attributes, name, owner, permissions, length, and modification time. The cache is a map from path name to responses, which we partially store in shared memory. Cache invalidation happens on conflicting file system operations: create, append, write, rm, rename, mkdirs, chmod, and chown.

	Throughput		Latency	
Server Application	queries per ms	CoLoRS /App	in ms	App/ CoLoRS
Cassandra	249.50	19	0.12	3
HDFS	12.03	20	0.19	3

Table 7.6: End-to-end performance for Cassandra and HDFS with caching. The third and fifth column show number of times improvement due to CoLoRS for throughput and latency, respectively.

Figure 7.4 presents the timing data for Cassandra and CoLoRS (left graph) and HDFS and CoLoRS (right graph). The x-axis is message size and the y-axis is time in seconds. We use this data to compute latency and throughput, which we summarize in Table 7.6. Columns 2–3 show transaction rate (per millisecond) while Columns 4–5 present response time (in ms). We use one cache warmup iteration followed by 10 iterations during each of which we vary the query result size between 1 and 1024 entries. In each column group, we report measurements for the server without CoLoRS and the relative improvement due to CoLoRS. For cache-enabled Cassandra, CoLoRS improves transaction rate by 19 times and reduces response time by 3 times. For cache-enabled HDFS, CoLoRS improves transaction rate by 20 times and decreases response time by 3 times.

7.3.5 CoLoRS Overhead

To implement CoLoRS, we virtualize components of Java and Python runtimes. This includes standard libraries, object field access, synchronization, method dispatch,

Bench- mark	Python 3.1 time (s)	CoLoRS 3.1 % OHead	Python 2.6 % Impr
binary-trees	6.79	3.39	-0.44
fannkuch	1.97	4.57	24.68
mandelbrot	15.32	7.18	66.52
meteor-contest	2.25	1.78	32.35
n-body	8.67	2.08	7.04
spectral-norm	14.31	5.73	18.85
pybench	3.92	5.20	1.18
pystone	4.09	5.87	12.98
Average	7.17	4.48	20.40

Table 7.7: The overhead of CoLoRS support for Python (and for the use of Python v3.1 over v2.6). Column 2 is execution time in seconds. Column 3 shows the percent degradation due to CoLoRS. Column 4 shows the percent improvement in performance when we use Python 2.6 (over 3.1).

interpreter, dynamic compiler, allocation, and GC. Doing so provides transparency, but introduces execution time overhead. To evaluate this overhead, we compare unmodified release versions of Python 3.1 and Java 1.6 with their CoLoRS counterparts.

Table 7.7 shows Python results. In Column 2, we report per-benchmark execution times for unmodified Python 3.1. Next, in Column 3, we present the CoLoRS overhead – percentage increase in execution times relative to Column 2. Across our benchmarks, the average CoLoRS overhead is 4%. Note that scripting languages are not concerned with enabling high-performance (they are interpreted and much slower than statically compiled code).

Table 7.8 shows the Java results. For each benchmark, we report its heap size and execution time (for DaCapo – the top 11 benchmarks) or throughput (for SPECjbb),

Bench- mark	Heap Size	ET or TP	CoLoRS Support % Overhead
antlr	7	2.40	8.4
bloat	28	6.34	6.3
chart	42	6.19	6.1
eclipse	115	24.54	4.7
fop	28	2.11	7.7
hsqldb	280	3.35	3.6
jython	3	8.35	4.5
luindex	7	7.50	9.0
lusearch	45	4.25	1.4
pmd	56	6.92	8.6
xalan	105	5.97	-0.6
jbb’00	900	112726	5.3
jbb’05	900	54066	1.3

Table 7.8: The overhead of CoLoRS runtime support for Java. Column 3 is execution time (ET) in seconds for all but jbb’00 and jbb’05 for which we report throughput (TP). Column 4 shows the percent degradation due to CoLoRS.

and percentage CoLoRS overhead (Column 4). Across the benchmarks, the average CoLoRS overhead is 5%.

7.3.6 Sockets vs. Shared Memory

We also investigate the relative performance of shared-memory-based transport (SMTx) and local-socket-based transport (LSTx). This enables us to determine how much performance improvement is due to the use of shared memory versus of sockets and due to avoiding object serialization.

In this experiment, we extend the Thrift RPC framework for Java with SMTx and compare it with the LSTx already built into Thrift (using our microbenchmarks described in Section 7.3.2). We have implemented SMTx in Thrift on top of a bidirectional FIFO channel in a shared memory segment and POSIX mutexes/conditions. We focus on Java and Thrift here because of their high-performance characteristics.

We observe that Thrift over LSTx attains better throughput – the improvement ranges from 1.7x (for the integer payload) to 3.2x (for 4-level binary trees) and averages at 2.7x. At the same time, Thrift over SMTx has lower latency for small messages (by up to 29% for the integer payload) and higher latency for larger payloads (by up to 0.8x for 4-level binary trees), while averaging at 9% lower latency than Thrift over LSTx.

The fact that Thrift/LSTx achieves better overall communication performance than Thrift/SMTx can be attributed to a more efficient sockets implementation (in the kernel) than our shared-memory queue implementation (in user-land). In the kernel, there is more control over memory mapping and thread scheduling, both of which can be used to optimize sockets implementation (e.g. to reduce the amount of copying and thread context switching).

Based on this experiment, we can conclude that CoLoRS improves throughput and latency because it avoids serialization and not because it uses shared memory instead of sockets.

7.3.7 Results Summary

CoLoRS can improve communication performance significantly when runtimes executing interoperating components (potentially written in different languages) are co-located on the same physical system, compared to extant type-safe cross-language RPCs (latency 2–113 times and throughput 8–102 times). In systems with short request processing times (e.g. servers with caches) this improvement can translate to large end-to-end performance gains (19–20x for transaction rates and 3x for response times). As more and more components are co-located on multi-cores and caches become prevalent in servers, object sharing systems like CoLoRS have a growing potential for increasing performance of multi-component, multi-language systems.

7.4 C/C++ Support for CoLoRS

The main challenges in implementing CoLoRS for unmanaged programming languages, such as C/C++, that provide no language/runtime support for automatic memory management, threading, concurrency, synchronization, and type reflection, are the following:

- Guaranteeing type-safety for the objects in the shared memory while preserving pointer arithmetic, unsafe memory accesses, and unchecked type casts (useful for systems programming) in the private memory.

- Providing transparent access to private and shared objects in terms of field load/store, virtual dispatch, pointer and operator usage, builtin primitive types, and the standard library (STL, at least for string, map, list, and set).

- Extending the language runtime to include sufficient reflective information to enable implementation of class mapping, recursive object graph traversal/copying, and dynamic field offset translation.

- Extending the memory management subsystem with support for techniques typically used by modern managed runtimes, such as multithreaded allocation in TLABs, precise root scanning, concurrent pauseless garbage collection, etc.

- Providing monitor synchronization semantics in a form of a library and reconciling the CoLoRS memory model with the C/C++ memory model.

To address these issues, one can either modify the C/C++ compiler (e.g. gcc) or use source-to-source C/C++ code translation. We have taken the latter approach because it is simpler and provides portability across the C/C++ compilers. Before compiling a C/C++ program that uses shared memory to an executable binary, CoLoRS translates the program source code into its CoLoRS-safe equivalent (by using pointer/field wrappers, templates, and operator overloading).

7.4.1 Type Safety

To guarantee type- and memory-safety in the shared memory, CoLoRS intercepts all pointer-based memory accesses as well as pointer arithmetic. Unsafe pointers to the objects in the shared memory (produced for instance via pointer arithmetic or arbitrary type casts) are disallowed. As soon as a program creates such an unsafe shared pointer, an exception is thrown. Although shared pointers cannot be manipulated, unsafe pointers in the private memory are allowed and normal pointer arithmetic still works for them.

CoLoRS achieves memory safety by wrapping all pointers in the program source code in an object and redefining pointer-related operators. Thus, the system can detect all pointer manipulations and check that all pointers to shared objects are correct and that there are no shared-to-private pointers.

For each local/global variable, function argument, and object field whose type is `T*` we change the type to `xptr<T>`. The `xptr` template class mimics pointer behavior by operator overloading and implicit type conversion.

7.4.2 Transparency

Shared memory objects can be accessed only via pointers. Although C/C++ supports both pointer and value types, shared data cannot be used in a non-pointer context. For example, assuming that we have pointer `p` of type `A*`, expressions like `p->name`, `p->print()` are allowed, however expressions like `*p`, `p[2]` are disallowed. The reason is that a non-pointer context requires copying, and copying data from the shared memory to private memory is not supported because of the potential ambiguity of type mapping in CoLoRS (one-to-many mapping relationship, e.g. shared class `integer` maps to C/C++ int/char/long etc.) Another reason is that even if copying to private memory worked, updates like `p[2].name = NULL;`, would be lost as they would execute on private copies.

C++ references are not supported because obtaining a C++ reference to a shared object requires going through a non-pointer context, e.g. `A &r = *p;`. Supporting references would require substantial parsing/translation effort (to optimize away the non-pointer context in cases where no actual copying is meant by the programmer).

CoLoRS uses C++ exceptions to signal errors (*shared_memory_exception* is thrown on error). In C++, type hierarchy does not have a single root (unlike *java.lang.Object* in Java) and therefore the CoLoRS API relies on templates (the CoLoRS API functions are generated for the types that actually use them instead of having one implementation for the root type).

Share-able classes may contain only builtin types and pointers. Integer builtins (`char`, `short`, `int`, `long`, `long long`, and their `unsigned` variations) map to `integer`. Floating point builtins (`float`, `double`, and `long double`) map to `float`. C++ `bool` maps to `boolean`. Additional predefined mappings exist for STL `string`, `list`, `set`, and `map`. Below, we show an example share-able C++ class.

```
class Person {
  string *name;
  double salary;
  Person *manager;
};
```

Fields `name` and `manager` are pointers because they are non-scalar.

C++ uses namespaces for identifier scope management. CoLoRS type mapping relies on fully-qualified class names. In C++, CoLoRS builds fully-qualified class names using subsequent nested namespaces and dot as a separator.

7.4.3 Programming Interface

The CoLoRS C++ API is equivalent to the CoLoRS Java API (i.e. it supports repositories, copying to shared memory, direct allocation, pointer testing for being private/shared, and reflection). CoLoRS adds the monitor synchronization API (lock, unlock, wait, notify, and notify_all) because C++ lacks support for monitors in the language. The memory model imposed by the synchronization API is consistent with the C++ memory model (i.e. semantics is given only to properly-synchronized programs). The CoLoRS API uses templates with generic code for user-defined classes and template specializations for builtins.

7.4.4 Type Reflection

We extend the C++ runtime with reflective information because the C++ RTTI does not support inspecting field types, inheritance hierarchy, and member functions. With source-to-source translation, gathering reflection data is straightforward. While parsing/translating a class definition, we collect typing information and emit it as soon as the class is processed. For each field in a class we record its name, type, offset, and size (the last two are necessary because when copying a private object to the shared memory we need to read it field by field).

7.4.5 Pointers, Fields, and Pointers to Members

We virtualize pointers, fields, and pointers to members, using three main wrappers: `xptr` for pointers, `xfld` for fields, and `xoff` for pointers to members. The `xptr` template class looks as follows.

```
template <class T>
class xptr {
  T* forward;
  long index;
};
```

The class contains the actual pointer (forward) and its index in the table of `xptr` objects (this table is used for garbage collection in the shared memory). Thus, wrapped pointers are twice bigger than regular C/C++ pointers.

Since C/C++ supports pointers to pointers, `xptr` objects might form a chain. For example `A** p`, becomes `xptr<xptr<A>>`. The `xptr` class overloads pointer arithmetic and implicit conversion to and from `T*`.

Each non-static field is wrapped in `xfld`, a template class parameterized by field type (T), field index (I), and the enclosing class type (H), as shown below.

```
template <class T, int I, class H>
class xfld<T, I, H> {
  T value;
};
```

For our example `Person` class, the following field wrapping is generated by the translator:

```
class Person {
 xfld<xptr<xlang_string>, 0, Person> name;
 xfld<double, 1, Person> salary;
 xfld<xptr<Person>, 2, Person> manager;
};
```

Note that in this case there is no space overhead (the actual field value is the only field in the `xfld` wrapper).

Template parameters `<T,I,H>` are necessary to transparently implement field access. Suppose that we execute `p->salary = 100;` on a shared pointer `p`. This invokes the assignment operator in the `xfld` wrapper. The implementation of this operator must compute the actual receiver (shared object address), get the shared type of the receiver, map this shared type to a local type, compute the shared field offset based on the mapping, and do the actual field store. The holder (H) and index (I) parameters are needed to compute the receiver (`p` - local_offset(`salary`)) as well as to perform type mapping.

The `xfld` class overloads a number of operators to emulate regular field behavior (arithmetic/comparison operators to support numeric/pointer fields, conversion to and from `T`, assignment operators, etc.)

We use the `xoff` wrapper to support C++ pointers to fields in the shared memory. The wrapper contains only the field index, as shown below.

```
template <class M, class T>
class xoff {
  int index;
};
```

During translation, pointers to fields are replaced with `xoff`, e.g. `int A::*p` becomes `xoff<int, A> p`. For transparency, the `xptr` class defines the `->*` operator that takes `xoff` (`xoff<int, A>` in this case) as a parameter and returns `xfld`. The `xoff` class has an implicit conversion from all potential pointers to fields (in this case from `xfld<int, I, A> A::*p` for all I used in A). Since C++ does not allow the dot operator to be overloaded, we translate expressions like `a.*p` into `(&a)->*p`.

7.4.6 Class Mapping and Loading

We maintain a mapping from shared classes to private classes as well as mapping from full private class names to private classes. The latter is needed whenever an unmapped shared class is encountered (to find a private class for it by name). The former is used during each field access/virtual call for offset mapping/dynamic dispatch emulation.

We use STL `map` to implement both mappings (by shared class and by full name). Each entry in the by-shared-class mapping contains a private class pointer and a vector for field offset mapping. This mapping is built dynamically, as classes are encountered in the shared memory.

For performance reasons, each thread caches recently-used mappings in thread-local (POSIX TLS) partial copies of the two global maps. Thanks to this, mapping can be done without synchronization in the common case. The global maps are consulted only if the lookup fails in the cached maps.

7.4.7 Garbage Collection

CoLoRS GC requires each attached VM/MRE to report all pointers (roots) to the shared memory on request. C/C++ does not have any mechanism for precise stack scanning, safepoints, and locating pointers in the heap. However, we can reuse the `xptr` wrappers for finding all shared roots. To do that, we introduce the root table – all `xptr` instances present in the private memory that wrap shared pointers are registered in the root table. Threads bulk-allocate multiple root table entries to reduce synchronization overhead. On construction, an `xptr` object registers itself in the root table, and on destruction it deregisters itself. Thus, roots can be reported any time by scanning the root table. The system uses a write barrier when registering a shared pointer to ensure that all pointers are captured even if a sequential scan (over the root table) misses a root. The root dump is fully concurrent, we do not need to stop any threads (CoLoRS GC imposes no pauses). This is consistent with the C/C++ programming model, where there are no asynchronously-triggered pauses.

7.4.8 Virtual Dispatch

The C++ virtual call mechanism is based on a virtual table pointer present in each object whose class has a virtual function. The C++ ABI mandates this pointer to be the first word in an object. Dynamic dispatch in C++ first fetches the virtual table pointer, then loads the function address from the table, and finally calls the function. We cannot use it directly on shared objects. However, we can reuse it by introducing proxy objects. Proxy objects are created based on private classes. We first map a shared class to a private class, then create a proxy object, perform a C++ native dispatch on the proxy, and finally statically call the right function for the original (shared) receiver. For example, suppose we have class `A` which defines a single virtual function `f`, as shown in the code fragment below. The translator replaces class `A` with its instrumented version that has two additional functions.

```
// original code:
class A {
public:
  virtual void f(int a) {
    /* f code */
  }
};

// generated code:
class A {
public:
 virtual void f_xlang(int a) {
   /* f code */
 }
 virtual void f_xlang2(int a) {
    A* xlang_recv = xlang_fix_receiver(this);
    xlang_recv->A::f_xlang(a);
 }
 inline void f(int a) {
  if (not_in_shared_memory(this)) {
    f_xlang(a);
  } else {
    XLangVTBLWrapper xlang_wrapper;
    A* xlang_recv = xlang_receiver(this,
       &xlang_wrapper);
    xlang_recv->f_xlang2(a);
  }
};
```

Note that the original function `f` is no longer virtual, it is statically-bound and inlined. In that function, we first check if we have to deal with a shared object. If not, we simply proceed to a regular C++ dynamic dispatch on the current receiver. Otherwise, we create a proxy object on the stack. This is accomplished by the `xlang_receiver` function, which initializes both fields of the proxy object.

```
class XLangVTBLWrapper { // proxy
public:
  void *vtbl_pointer;
  void *shared_object;
};
template <class T>
T* xlang_receiver(T *t, XLangVTBLWrapper *w) {
  w->shared_object = (void*)t; // receiver
  // map shared type of t to local type lc
  w->vtbl_pointer = lc->vtbl_pointer;
  return (T*)w; // return the wrapper
}
```

The shared receiver (`t`) is stored in the proxy object for future use. At the same time, we map the shared class of `t` to a private class. This private class is used to set up the virtual table pointer (the first word) of the proxy object.

Once the proxy is initialized, we perform a normal C++ dispatch on it. After the dispatch, we end up in the `f_xlang2` function, where we restore the previously-saved shared receiver and call statically the function that corresponds to the function we ended up in.

Since most functions in C++ programs are static (non-virtual) we perform the above transformation only for classes that have virtual functions. Determining if a function is virtual requires walking up the class hierarchy, therefore to simplify the translator we conservatively find all virtual functions by name – if a specific name occurred earlier in the context of a virtual function we assume that the function is virtual.

To obtain a virtual table pointer for a private class (we need it to set up a proxy) in a portable way, we create an object instance for each class with virtuals, before the program starts. To do that, we emit an empty constructor chain (for inheritance and membership relationships) and call this empty constructor chain to silently (i.e. without any side effects) instantiate and delete an object. We save the first word as a virtual table pointer for later use.

7.4.9 Standard Libraries

We virtualize STL `string`, `list`, `set`, and `map` so that shared and private instances of these classes can be used transparently. Our general approach is to implement a wrapper class with the same API as the original class. Each API function first checks whether a private or local implementation of the function should be used. For the private case, we delegate to the wrapped instance. For instance, for `std::string` the wrapper class is the following.

```
class xlang_string {
  std::string value;
public:
  size_t size() const {
    if (not_in_shared_memory(this))
      return value.size();
    // shared implementation
  } ...
};
```

The translator replaces all `std::string` occurrences in the program source with `xlang_string`.

7.4.10 Implementation Details

We use JavaCC, an open-source recursive-descent parser generator for Java that supports variable look-ahead LL(k) grammars. We modify a publicly available ANSI C/C++ grammar for JavaCC to implement a single-pass translator. The translator builds no abstract syntax tree and mostly copies input to output. Occasionally, a sequence of tokens is buffered and processed together, for example to emit pointer/field wrappers. While parsing the input, the translator gathers reflective information about classes/fields, which is then emitted once a particular class gets fully parsed. The translator does not perform any syntactic/semantic correctness checks (we assume that the input code compiles correctly because this can be easily checked before the translation begins).

For efficient direct allocation in the shared memory and fast object graph copying, each private class has a pointer to a shared class that fully matches the private class. This avoids repetitive class comparison/lookup.

The shared memory segment is mapped at a pre-defined address in virtual memory. Thus, border checks are inlined comparisons with constants (macros) that the C/C++ compiler can regroup and optimize away.

Bench-mark	CoLoRS		CORBA		Proto.-Buff.		Thrift	
	Thr. [call/ms]	Lat. ms	Thr. [rel.]	Lat. [rel.]	Thr. [rel.]	Lat. [rel.]	Thr. [rel.]	Lat. [rel.]
boolean	18.47	19.97	1.69	5.51	3.04	2.19	5.59	3.21
integer	6.86	28.44	3.07	3.90	1.34	1.55	2.35	2.22
float	5.98	35.40	2.97	3.05	1.46	1.35	2.16	1.76
string	1.99	16.70	2.44	6.93	2.99	3.43	2.37	3.93
1-tree	0.61	29.30	1.54	6.72	1.46	2.55	1.96	2.48
2-tree	0.24	25.00	1.97	5.97	1.70	2.75	2.51	2.95
3-tree	0.11	19.92	2.16	11.25	1.85	4.33	2.78	4.63
4-tree	0.05	43.75	2.47	11.54	2.09	2.59	3.10	3.21
average	4.29	27.31	2.29	6.86	1.99	2.59	2.85	3.05

Table 7.9: Microbenchmark performance for CoLoRS, CORBA, Protocol Buffers, and Thrift. Columns 2 and 3 show absolute throughput (calls per millisecond) and latency (in milliseconds) for CoLoRS. Columns 4–9 show relative throughput degradation and relative latency increase compared to CoLoRS (we report number of times degradation/increase).

7.4.11 Experimental Evaluation

We compare the performance of CORBA, Protocol Buffers, and Thrift with the RPC implemented on top of CoLoRS. We use a C++ client and a Java server and employ the same communication microbenchmarks and methodology as in our Python-Java experiments. For the CORBA C++ client we use omniORB 4.1.4. We extend Protocol Buffers with the TCP/IP transport and send the serialized messages using the TCP_NODELAY flag.

Table 7.9 summarizes the results. We report per-microbenchmark throughput and latency: absolute values for CoLoRS and relative values for CORBA, Protocol Buffers, and Thrift (number of times throughput degradation and number of times latency increase compared to CoLoRS). On average, CoLoRS throughput is better by 2x to 3x, and CoLoRS latency is shorter by 3x to 7x. Among all RPCs, CORBA has the highest latency and Thrift has the lowest throughput.

7.5 Related Work

CoLoRS is unique in that it supports type-safe, transparent, and direct object sharing via shared memory between managed runtimes for different static/dynamic object-oriented languages. To enable this, CoLoRS defines a language-neutral object/memory model as well as a synchronization mechanism and concurrent/on-the-fly GC, all designed specifically for multi-VM cross-language object sharing.

CoLoRS takes a top-down approach to object sharing. That is, we assume full isolation between the runtimes via operating system (OS) process semantics and provide a mechanism for object sharing within this context. Several previous systems [53, 10, 65, 115] took a bottom-up approach by executing multiple applications in a single OS process and providing software-based isolation between them.

State-of-the-art systems that support type-safe, cross-language communication for OO languages, such as OMG CORBA [50], Apache Thrift [143], Google Protocol Buffers [129], SOAP, and REST, target distributed systems and rely on message-passing and data serialization. CoLoRS differs from these systems in that it targets co-location and transparent shared memory (as opposed to explicit message passing). Although one can use CoLoRS to implement an efficient cross-language RPC for the co-located case (similar in spirit to LRPC [26]), CoLoRS is more general than RPC systems and differs from them in terms of both architecture and programming model.

XMem [160] provides direct object sharing between JVMs. XMem also takes a top-down and transparent approach, but does not support sharing between heterogeneous languages and requires global synchronization across runtimes (which CoLoRS avoids) for such operations as garbage collection, class loading, shared memory attach/detach, and communication channel establishment.

Systems supporting communication between isolated tasks within a single-language, single-process runtime include Erlang [7], KaffeOS [10], MVM [53], Alta [11], GVM [11], and J-Kernel [157]. These systems take a bottom-up approach which provides weaker isolation (i.e. weaker protection guarantees than the CoLoRS approach) and is more complex to implement. Unlike CoLoRS, they replicate OS mechanisms within a single OS process instead of leveraging existing hardware-assisted inter-process isolation.

Language-based operating systems also provide mechanisms for communication and interoperation between processes [138, 65, 89, 74, 96, 63, 27, 170, 94]. Such systems typically implement support for light-weight processes that share a single address space and provide compiler support to guarantee type and control safety within and between processes. To facilitate the latter, these systems require that the components (processes/tasks) be written in the same safe/checkable language. In addition, since CoLoRS is not an operating system, it is significantly simpler.

Some concurrent languages provide direct support for inter-process communication between light-weight processes [8, 117, 63] written in the same language. The key difference between these systems and CoLoRS is that they employ share-nothing semantics for message-based communication whereas CoLoRS provides support for direct object sharing when runtimes are co-located on the same physical machine.

CoLoRS is also distinct from distributed shared memory and single system image runtimes for clusters such as MultiJav [41], cJVM [6], JESSICA [106], Split-C [51], and UPC [64]. In contrast to them, CoLoRS provides a uniform cost for accessing all objects (private and shared) and does not target distributed computing. These systems provide sharing between code written in the same language, and focus on guaranteeing memory consistency and cache coherence for concurrent access to objects across multiple machines.

7.6 Summary and Conclusions

CoLoRS provides cross-language, cross-runtime, type-safe shared memory for co-located MREs. CoLoRS defines a language-neutral object/class/memory model for static and dynamic OO languages, as well as an on-the-fly, concurrent GC and a monitor synchronization mechanism both adapted and extended to support language- and runtime-independent object sharing.

We implement and evaluate CoLoRS within runtimes for Python and Java. CoLoRS imposes low overhead when there is no use of shared memory (4% for Python and 5% for Java) due to virtualization of runtime services and libraries. An important use case for CoLoRS is improving the performance of RPC protocols in the co-located case. We have found that for microbenchmarks CoLoRS increases throughput by 8–102 times and reduces latency by 2–113 times. CoLoRS improves the performance for the cache-enabled Cassandra database and HDFS by 19–20 times for throughput and 3 times for latency. In summary, CoLoRS enables type-safe, object sharing across OO languages in a transparent and efficient way.

The text of this chapter is in part a reprint of the material as it appears in [164].

Chapter 8

Conclusion

In this dissertation, we investigate techniques for improving memory management in multi-language, multi-runtime systems that co-locate multiple isolated components on multi-core shared-memory architectures. Such systems are becoming increasingly common because of a number of reasons. First, in order to enhance programmer productivity, developers more and more often use high-level, type-safe, object-oriented, and portable programming languages, and execute applications within managed runtimes. Second, to manage software complexity, architects typically divide systems into multiple components, which execute in separate runtimes for resource and fault isolation. Third, to reduce development time, each component is usually implemented using the programing language that is most suitable to its functionality, dependencies, and performance requirements. Finally, administrators increasingly co-locate multiple components to utilize the resources of multi-core architectures and reduce the cross-component communication overhead.

The goal of our work is to improve the performance and programming model of multi-language, multi-runtime systems deployed on multi-core machines by leveraging OS support for memory management. We investigate new techniques for both intra-runtime (object allocation and garbage collection) and cross-runtime (object sharing, message passing, and remote procedure calls) memory management. More specifically, we design, implement, and evaluate MRE extensions that enable better coordination across the memory management subsystems in the OS kernel and in an MRE. These extensions are described in detail in Chapters 3–7 and leverage OS support for:

- **Virtual memory.** Since unreachable objects form large clusters in the heap, they can be effectively managed at the granularity of virtual pages instead of individual objects. To improve the performance of intra-runtime memory management, we develop two collectors, MC and YP, that exploit this statistical property and use the OS virtual memory subsystem. MC leverages page remapping operations to implement partial compaction in virtual memory. It unmaps individual empty pages and maps them as a new contiguous region in virtual memory. As MC moves dead space instead of live space, it avoids costly object copying and pointer updates, which results in higher throughput and shorter pauses. YP leverages the kernel page reference bits to estimate the percentage of the heap that is reclaimable and guide the GC triggering mechanism accordingly to avoid unproductive collections. This allows to reduce the number of GCs and thus increase throughput and MMU.

- **Shared libraries.** We design and implement a portable and lightweight shared library that enables integration of parallel and concurrent GC into existing or new managed runtimes. The library decouples the GC implementation from MRE internals via a C interface. It improves intra-runtime memory management in two ways. First, by increasing MRE modularity, the library simplifies the programming model for MRE developers. Second, by providing an efficient, optimized GC implementation, it improves MRE performance.

- **Shared memory.** For cross-runtime memory management, we develop type-safe, transparent object sharing that uses OS shared memory segments and the associated OS inter-process synchronization primitives. We investigate cross-runtime sharing in a single-language (Java) and multi-language (Java, Python, C++) setting. In both cases we virtualize such runtime services and components as object allocation, GC, field access, method dispatch, monitor synchronization, class loading, and system libraries. We find that direct object sharing increases throughput and decreases latency by up to several orders of magnitude compared to state-of-the-art type-safe cross-runtime communication protocols based on remote method invocation and messaging that require object serialization. In addition to improving performance, shared memory also enriches the programming model by adding the ability of sharing that has not been available in managed runtimes to date and that is more natural than explicit message passing for many applications.

Detailed empirical evaluation of our MRE extensions shows that they enable performance improvements both in intra-runtime (GC) and cross-runtime (inter-MRE communication) memory management. In addition, they enhance the programming model for both application and MRE developers through new communication primitives and by simplifying MRE implementation through separation of concerns, respectively.

8.1 Contributions and Impact

In this section, we summarize our main contributions and discuss their impact. Our primary contribution is improving performance and programming model of state-of-the-art memory management within and across managed runtimes by better cross-layer coordination, specifically OS support for MREs. Other contributions that we make in this dissertation include reducing complexity and increasing modularity of MREs, exploiting statistical properties exhibited by programs at runtime, developing better techniques for garbage collection and direct object sharing, as well as leveraging recent hardware and software trends (multi-cores, co-location, large 64-bit address spaces, etc.) to improve the efficiency of memory management systems.

The results of our research have appeared in the proceedings of high-impact-factor peer-reviewed conferences such as PLDI, ASPLOS, and OOPSLA. Type-safe, transparent shared memory across different, static and dynamic languages has never been investigated in the literature before. Besides their scientific impact, our contributions have a significant practical value. MREs and type-safe languages with automatic memory management have become the major development platform for both applications and systems. A wide array of software technologies today, ranging from deskside applications to enterprise middleware, rely on MREs for object-oriented languages, garbage collection, and type-safe RPCs. Improving performance and programming model of such systems deployed in production settings has the potential to impact many users and developers.

Below we describe our key contributions in more detail and explain how they relate to the specific techniques and systems discussed in Chapters 3–7.

- **Improved cross-layer interaction.** We develop new ways of improving the integration of MREs with the underlying OS and making memory management in MREs OS-aware. Our results demonstrate that MREs can significantly benefit from more cooperative interaction with the lower-level layers of the software/hardware stack, while maintaining standard and portable MRE-OS interfaces (e.g. system calls, kernel modules, shared libraries, IPC mechanisms, and shared memory).

We identify new uses for the OS virtual memory support in MREs: MC (Chapter 3) uses page (re)mapping for efficient compaction, YP (Chapter 4) leverages the page reference bits for accurate yield prediction, and XMem (Chapter 6) exploits the level of indirection provided by virtual memory for double class mapping. XMem and CoLoRS (Chapter 7) are the first MRE systems described in the literature that integrate OS support for shared memory and inter-process synchronization into a managed runtime for the purpose of type-safe object sharing and coordination across different OS processes. GaS (Chapter 5) is currently the only shared library with C/C++ linkage that encapsulates a concurrent, on-the-fly GC for GC-cooperative runtimes.

- **Significant performance increase.** We contribute several system and algorithmic techniques that reduce the overhead imposed by parallel and concurrent GC used in state-of-the-art MREs. Our experimental evaluation shows that these contributions enable significant improvements in program execution time and/or GC pause times (responsiveness). These performance gains are due to low-cost page-based virtual compaction (MC), predicting and skipping unproductive collections (YP), and integrating an on-the-fly low-pause library-based GC into an

MRE (GaS). To date, page reference bits have not been used to optimize GC triggering (YP is the first system to leverage them in this context).

In addition, we optimize RPC performance by introducing cross-MRE and cross-language shared memory to avoid data structure serialization and copying in the co-located case. We observe orders of magnitude improvements in throughput and latency which translate to significant end-to-end performance gains. Presently, CoLoRS is the only system that can speed up local RPC across different type-safe languages.

- **Enhanced programming model.** To date, managed runtimes for safe languages have supported only message passing and RPC as means of cross-runtime communication. We enrich this programming model by providing the abstraction of type-safe shared memory as an alternative. Cross-MRE object sharing has not been investigated before and general-purpose languages have lagged behind the OS IPC in the scope of supported primitives. Shared memory can improve performance in the co-located case as well as it is a more natural communication mechanism and a system model for certain applications. One potential practical use case is improving communication performance for cross-language RPC in systems like backend servers at Google and Facebook that often co-locate different components. Another possible application are server-side systems such as the Oracle database [119] that are increasingly written in safe languages and whose architecture comprises a set of isolated OS processes and a shared memory segment. Multi-tiered enterprise web applications, which use multiple languages (e.g. PHP for the presentation layer, and Java/C++ for the database layer) and run in independent runtimes can benefit from efficient local RPCs. High-performance computing (HPC) systems using OO languages (C++, Java, etc.) and deployed on clusters of multi-core machines can use object sharing as a lightweight message passing replacement for co-located isolated worker processes.

Our contributions also improve the programming model for MRE developers. Memory management is one of the most complex subsystems in managed runtimes. Providing state-of-the-art on-the-fly GC, notorious for its implementation complexity, as a reusable library (GaS) reduces the development effort required for building new or improving existing MREs.

- **New memory management techniques and algorithms.** To enable better cross-layer memory management in the OS and MREs we developed a number of new GC techniques. We designed them specifically for more cooperative GC-OS interaction. This involved adapting extant algorithms and designing new ones, removing dependencies on runtime services, as well as decoupling and abstracting away the GC internals. Thus far, our GC systems have been used by researches from Zurich, Switzerland, at the University of Salzburg and University of Texas at Austin, as well as at AMD Operating System Research Center in Dornach, Germany.

MC is the first nearly-one-phase compactor (other GCs that implement compaction have two or more phases) and uses one of the simplest algorithms (avoiding object moving and pointer adjustment), which is equally-easy to employ in both stop-the-world and concurrent GCs. XMem and CoLoRS use parallel and concurrent GC that is adapted to work with isolated and scattered address spaces. GCs in these systems delegate the root dump operation to the currently attached managed runtimes. In addition, XMem and CoLoRS reduce the MRE-GC interface to a minimum to avoid tight coupling and the resulting lack of fault-tolerance. Both GaS and CoLoRS adapt the SATB GC algorithm by Doligez, Leroy and Gonthier to decouple GC from the runtime services. This adaptation introduces an additional phase and removes dependencies on global handshakes and per-thread write barrier buffers. To the best of our knowledge, this is the simplest and most decoupled on-the-fly GC published today. YP introduces a novel GC triggering mechanism based on page reference bits whose adaptive prediction is guided by the feedback from GC. YP is the first GC system that uses program reference behavior to optimize the GC frequency and timing.

- **Type-safe object sharing across languages.** When we started this dissertation work, object sharing systems were either limited to a single language (mostly Java) or to a single OS process (e.g. KaffeOS, MVM). In addition, the design of most such systems was based on the top-down approach, which is complex, provides weak software-only isolation, and duplicates OS cross-process resource protection. Distributed shared memory systems (e.g. cJVM, MultiJav) not only targeted one language but also focused only on optimizing the distributed protocols while ignoring co-location. At the same time, OS support for shared memory, despite having been standardized and used in production for decades, was neither exploited by managed runtimes nor exposed to application developers at the level of abstraction matching the programming language. In consequence, programs written in high-level languages had to rely on expensive message passing protocols and occasionally adjust the programming model to fit the available abstractions.

  Two our most important contributions, XMem and CoLoRS, significantly changed the landscape of type-safe, cross-runtime and cross-language communication. We took a different, bottom-up design approach and built lightweight shared memory for OO languages that reuses extant IPC facilities and strong OS inter-process isolation. CoLoRS is the first system that provides cross-language direct sharing for co-located runtime processes. It addresses a number of previouslyunexplored research questions and design tradeoffs pertaining to such aspects as language-neutral object model and memory model, efficient support for dynamic translation between language-specific and shared object layouts, type-safety in hybrid static and dynamic type systems, and decoupled but transparent GC and synchronization. CoLoRS is currently used by AppScale [44], a multi-language distributed cloud system, to optimize communication between intermittently co-located components.

- **Reduced system complexity.** While designing all the systems that we contribute herein, we strived to leverage OS support not only to improve performance and enhance the programming model but also to reduce the complexity of subsystems and services implemented by managed runtimes. A primary design goal of GaS is to simplify MREs by decoupling GC as a library that exposes a well-defined API. GaS adapts the SATB on-the-fly algorithm to avoid phase transitions, handshakes, and signal polling, all of which complicate the GC implementation. MC significantly simplifies concurrent compaction by moving (via remapping) dead space instead of relocating live objects and fixing pointers. XMem uses double page mapping to simplify dynamic class resolution and avoid introducing a level of indirection for dynamic dispatch and type reflection. CoLoRS leverages fast mutex implementations in modern OS (based on user-mode atomic operations) to simplify monitor implementation by obviating the need for lightweight locking.

YP exploits dead object clustering and the OS page replacement mechanism to avoid the complexity of heuristics used in extant systems and reduce the problem of prediction to counting not-recently-referenced pages.

- **Improved decoupling and modularity.** The MRE extensions that we contribute increase the MRE modularity by making the memory management subsystem loosely coupled with the runtime internals. GaS decouples on-the-fly GC from MREs via a simple C interface. The GaS library makes no assumptions about the object model, threading support, dynamic compiler, and heap management framework used by a managed runtime. Both GaS and CoLoRS remove the dependencies of the SATB GC on global safepoints and conditional multi-state write barriers. In addition, CoLoRS adapts the tightly-coupled lightweight and biased locking schemes in order to separate out the monitor implementation from the MRE. XMem and CoLoRS also modularize the GC by division of responsibility: MREs implement the root dump operation while tracing and sweeping is done by the shared memory server.

- **Better leverage of statistical properties of programs.** We develop and evaluate the effectiveness of new uses for the widely-known statistical observation that dead objects cluster together in the heap. This property underpins the design of MC and YP. MC exploits clustering to implement fast partial compaction with a high degree of defragmentation. YP leverages clustering to estimate the percentage of the heap occupied by dead objects by exploiting the fact that dead clusters are not referenced.

We optimize most mechanisms for the common case that we either establish experimentally or infer from well-known program properties. XMem and CoLoRS implement TLAB allocation that avoids free-list access and the associated synchronization overhead for small-to-medium objects. CoLoRS optimizes method instrumentation in the compiled code by assuming that most methods use only private data. CoLoRS hash-based monitors exploit the fact that there are few conflicts in the table. GaS GC assumes that most mutations happen to the recently-allocated objects. CoLoRS object model is designed around the assumption that dynamic field additions are relatively rare and thus most objects are allocated in one contiguous chunk.

- **Exploiting architectural advances to reduce design tradeoffs.** One of our research goals is to leverage recent hardware trends and extant OS support to eliminate or diminish the design tradeoffs present in state-of-the-art memory management systems. MC uses large 64-bit virtual address space to enable non-moving compaction and avoid the tradeoff between the cost of defragmentation and allocation speed. In addition, MC provides a simple concurrent compacting GC that does not synchronize with mutators, eliminating the system complexity vs. concurrent defragmentation tradeoff. XMem and CoLoRS leverage multi-core architectures and co-location to reduce the tradeoff between modularity/isolation and cross-component communication performance. GaS addresses the tight integration vs. GC performance tradeoff. YP enables better dynamic control over the space/time tradeoff in MREs that use GC.

- **Comprehensive experimental evaluation.** For each of our systems, we perform a comprehensive experimental evaluation based on standard community benchmarks and open-source applications. Our microbenchmarks are modeled after actual application behavior. We use a variety of metrics, including execution time, pause times, MMU, throughput, latency, and scalability. Our evaluation uses production-quality infrastructure: HotSpot JVM and cPython are the most widely-used, most efficient, and sophisticated runtimes for Java and Python available today. We compare our systems to state-of-the-art GCs and RPCs used in both research and production settings. Our methodology, metrics, and experimental setup reflect the best practices used in the memory management community.

- **Open-source implementation.** We contribute our implementations as open-source GPL projects available for download for free. The code base for MC, YP, and XMem has been already used by other researchers. Our implementations require standard, portable OS services and libraries and have been tested under various Linux distributions on several different architectures.

In summary, our contributions advance state-of-the-art in memory management primarily by improving performance and programming model, and secondarily by simplifying and modularizing the MRE architecture. They include novel systems and algorithmic techniques that have the potential to impact end users, application and MRE designers and developers, as well as programing language researchers.

8.2 Future Research Directions

In this section, we identify several avenues for future research work. Our contributions described in this dissertation motivate and facilitate designing and building new systems that further advance state-of-the-art in memory management, MREs, and beyond. We discuss a number of research directions that we believe are worth exploring based on our empirical results and observations as well as design and implementation intuition that we have gained while developing the systems we described in Chapters 3–7. We overview both extensions to our contributions and completely new research projects along with their potential impact.

Type-safe, transparent shared memory provided by CoLoRS can be a starting point for a number of different research paths. We identify and briefly overview the most interesting and promising ones below.

- **Distributed shared memory.** Cross-language, transparent, and lightweight sharing across a cluster of machines has never been investigated before. Previous work is limited to single-language systems, such as cJVM [6] and MultiJav [41] for Java. These single-system-image approaches are complex and heavyweight because they support whole-system-state sharing and thread migration.

- **Support for other object-oriented languages.** Other popular, type-safe, managed languages like Ruby, PHP, and C# may substantially benefit from shared memory as they are often used in enterprise web backends that are communication-intensive and rely on high-overhead RPC protocols. Support for additional languages would also verify the generality and usability of the CoLoRS object model.

- **Virtualization support and sharing across guest OSes.** CoLoRS may be extended to support object sharing across managed runtimes that execute in separate OS instances run in a virtualized environment. Vshmem [177] is a recent Linux extension that enables using shared memory segments across guest OSes executed on a single hypervisor. By leveraging the Vshmem API, CoLoRS can support cross-OS cross-runtime sharing.

- **Better fault-tolerance for critical sections.** An interesting research question is whether critical sections (delimited by monitor entry/exit) in the shared memory can tolerate arbitrary process failures. One possible approach to providing such fault-tolerance is using transactional memory. At the other end of the spectrum is limiting the programming model to atomic operations and dropping support for full monitor semantics.

- **Support for static fields and code.** CoLoRS currently provides only instance-field sharing but it can be extended with support for static fields with reasonable design and implementation effort. Another way to improve the system practicality, is to facilitate sharing of method code across languages. This can be accomplished either by extending CoLoRS or by providing additional tools, for instance for automating code generation via cross-language translation.

- **Fully transparent RPCs: automatic local/remote protocol selection.** Optimizing the performance of RPC protocols in the co-located case by using CoLoRS should not require any changes to the application code. To enable this, CoLoRS requires a co-location discovery mechanism (that identifies when communicating

VMs start/stop being co-located) to drive automatic selection between local and remote protocols.

We believe that leveraging OS support for MREs can be extended much further. Below, we describe a number of potential research directions that seem worth exploring and might lead to interesting results.

- **Fully-virtual compaction via multiple page mappings.** Instead of remapping empty virtual pages, we can remap multiple mostly-empty pages into a single page in a way that prevents overlapping of live objects in the address space.

- **Concurrent moving GC as an OS module to avoid the cost of signal handlers.** Certain concurrent GCs that compact the heap are expensive because they rely on page protection and frequent SEGV signals which require crossing the process-kernel boundary. Implementing part of the concurrent GC as a kernel module can eliminate this overhead.

- **Elimination of the overhead of virtual calls via virtual memory mapping.** The level of indirection provided by virtual memory can be used to devirtualize megamorphic call sites that cannot benefit from profiling, inline caches, and other dispatch optimizations used by managed runtimes.

- **Cooperative thread context switching in the kernel.** Extant OSes implement thread context switching without the knowledge of critical sections in the application code. This can lead to unproductive switches, for example, while a thread executes within a critical section. MREs could avoid this by making monitor synchronization cooperative with the kernel task scheduler.

- **Using page dirty bits to improve memory management.** Existing GC systems do not take advantage of page dirty bits, a hardware-assisted mechanism used by OSes to implement page replacement. Potential uses of dirty bits include write barrier elimination and efficient detection of heap properties at run-time.

- **Exploiting clustering and other statistics.** The empirical observation that dead/live objects cluster together may be exploited to a larger extent by MREs to implement different variants of page-based memory management. Other properties exhibited by modern programs, such as the fact that older objects have less fragmentation than younger ones [162], can also lead to better GC performance.

- **Decoupling other MRE components as simple libraries.** In addition to memory management, several other MRE subsystems may benefit from being engineered as reusable libraries, for example, dynamic compiler, interpreter, profiler, standard language libraries, etc.

- **GC for non-uniform memory architectures.** Many-core NUMA machines require a different approach to memory management, where the GC is aware of different memory access costs in the address space. Most extant GC algorithms and techniques assume uniform memory access and are thus unsuitable for NUMA.

In summary, our contributions open up several promising research opportunities in memory management and can be a foundation for further improvements in MRE performance, programming model, and architecture, as well as system modularity and MRE-OS coordination.

Bibliography

[1] D. Abuaiadh, Y. Ossia, E. Petrank, and U. Silbershtein. An efficient parallel heap compaction algorithm. In *OOPSLA*, 2004.

[2] Apache Cassandra Project. `http://cassandra.apache.org`.

[3] Apache Tomcat. `http://tomcat.apache.org`.

[4] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[5] A. W. Appel and K. Li. Virtual memory primitives for user programs. *ACM SIGPLAN Notices*, 26(4):96–107, 1991.

[6] Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *ICPP*, 1999.

[7] J. Armstrong. Erlang – a survey of the language and its industrial applications. In *9th ESIAP*, 1996.

[8] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1996.

[9] M. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[10] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *OSDI*, 2000.

[11] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java operating systems: Design and implementation. Technical report, Univ. of Utah, 1998.

[12] D. F. Bacon, P. Cheng, and V. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *POPL*, 2003.

[13] D. F. Bacon and V. Rajan. Concurrent cycle collection in reference counted systems. In *ECOOP*, 2001.

[14] H. G. Baker. Cache-conscious copying collection. In *OOPSLA*, 1991.

[15] H. G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, Mar. 1992.

[16] H. G. Baker. ‘Infant mortality’ and generational garbage collection. *ACM SIGPLAN Notices*, 28(4), Apr. 1993.

[17] H. G. Baker and C. E. Hewitt. The incremental garbage collection of processes. Technical report, MIT Press, 1977.

[18] D. Balfanz and L. Gong. Experience with secure multi-processing in Java. In *ICDCS*, 1998.

[19] K. Barabash, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko, and E. Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *TOPLAS*, 27(6):1097–1146, 2005.

[20] K. Barabash, Y. Ossia, and E. Petrank. Mostly concurrent garbage collection revisited. In *OOPSLA*, 2003.

[21] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In *PLDI*, 1993.

[22] J. F. Bartlett. Compacting garbage collection with ambiguous roots. Technical report, DEC Western Research Laboratory, 1988.

[23] J. F. Bartlett. Mostly-Copying garbage collection picks up generations and C++. Technical report, DEC Western Research Laboratory, 1989.

[24] BEA WebLogic Application Server. `http://www.bea.com`.

[25] O. Ben-Yitzhak, I. Goft, E. Kolodner, K. Kuiper, and V. Leikehman. An algorithm for parallel incremental compaction. In *ISMM*, 2002.

[26] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1), 1990.

[27] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. J. Eggers. Extensibility, safety and performance in the SPIN operating system. In *SOSP*, 1995.

[28] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *ICSE*, 2004.

[29] S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *PLDI*, 2002.

[30] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *OOPSLA*, 2003.

[31] H.-J. Boehm. Space efficient conservative garbage collection. In *PLDI*, 1993.

[32] H.-J. Boehm. Reducing garbage collector cache misses. In *ISMM*, 2000.

[33] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, 2008.

[34] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6), 1991.

[35] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[36] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric prediction of heap memory requirements. In *ISMM*, 2008.

[37] F. Breg and C. D. Polychronopoulos. Java virtual machine support for object serialization. In *Java Grande*, 2001.

[38] C. Bryce and C. Razafimahefa. An approach to safe object sharing. *SIGPLAN Not.*, 35(10), 2000.

[39] D. Buytaert, K. Venstermans, L. Eeckhout, and K. De Bosschere. GCH: Hints for triggering garbage collections. *THPEAC*, 1(1), 2007.

[40] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, 2005.

[41] X. Chen and V. H. Allan. MultiJav: A distributed shared memory system based on multiple Java virtual machines. In *PDPTA*, 1998.

[42] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, Nov. 1970.

[43] P. Cheng and G. Blelloch. A parallel, real-time garbage collector. In *PLDI*, 2001.

[44] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski. AppScale: Scalable and Open AppEngine Application Development and Deployment. In *ICCC*, 2009.

[45] T. W. Christopher. Reference count garbage collection. *SPE*, 14(6), 1984.

[46] C. Click, G. Tene, and M. Wolf. The Pauseless GC algorithm. In *VEE*, 2005.

[47] J. Cohen and A. Nicolau. Comparison of compacting algorithms for garbage collection. *TOPLAS*, 5(4):532–553, 1983.

[48] D. Cohn and S. Singh. Predicting lifetimes in dynamically allocated memory. In *ANIPS*, 1997.

[49] Computer Language Benchmarks Game. Language Performance Comparisons. `http://shootout.alioth.debian.org`.

[50] CORBA Specification. `http://www.omg.org`.

[51] D. E. Culler, A. C. Arpaci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. A. Yelick. Parallel programming in Split-C. In *SC*, 1993.

[52] G. Czajkowski. Application isolation in the Java virtual machine. In *OOPSLA*, 2000.

[53] G. Czajkowski and L. Daynes. Multitasking without compromise: A virtual machine evolution. In *OOPSLA*, 2001.

[54] The DaCapo Benchmark Suite. `http://dacapobench.org`.

[55] Dell Desktops and Servers. `http://www.dell.com`.

[56] A. Deshpande and D. Riehle. The total growth of open source. In *OSS*, 2008.

[57] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *ISMM*, 2004.

[58] A. Diwan, D. Tarditi, and J. E. B. Moss. Memory subsystem performance of programs using copying garbage collection. In *POPL*, 1994.

[59] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL*, 1994.

[60] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *POPL*, 1993.

[61] T. Domani, E. K. Kolodner, E. Lewis, E. E. Salant, K. Barabash, I. Lahan, Y. Levanoni, E. Petrank, and I. Yanorer. Implementing an on-the-fly garbage collector for Java. *SIGPLAN Not.*, 36(1), 2001.

[62] T. Domani, E. K. Kolodner, and E. Petrank. A generational on-the-fly garbage collector for Java. *SIGPLAN Not.*, 35(5), 2000.

[63] S. Dorward, R. Pike, D. L. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. Inferno. In *COMPCON*, 1997.

[64] T. El-Ghazawi, W. Carlson, and J. Draper. UPC Language Specifications V, 2001. `http://upc.gwu.edu`.

[65] M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*, 2006.

[66] R. T. Fielding. Architectural styles and the design of network-based software architectures. Technical report, Univ. of California, Irvine, 2000.

[67] C. Flood, D. Detlefs, N. Shavit, and C. Zhang. Parallel garbage collection for shared memory multiprocessors. In *USENIX JVM*, 2001.

[68] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice (Jini Series)*. Pearson Education, 1999.

[69] The GNU Compiler for the Java Programming Language. `http://gcc.gnu.org/java`.

[70] N. Geoffray, G. Thomas, C. Clément, and B. Folliot. A lazy developer approach: Building a JVM with third party software. In *PPPJ*, 2008.

[71] N. Geoffray, G. Thomas, J.Lawall, G. Muller, and B. Folliot. VMKit: a Substrate for Managed Runtime Environments. In *VEE*, 2010.

[72] P. Gepner and M. Kowalik. Multi-core processors: New way to achieve high system performance. In *PARELEC*, 2006.

[73] GNU Classpath. `http://www.gnu.org/software/classpath`.

[74] M. Golm, M. Felser, C. Wawersich, and J. Kleinoder. The JX operating system. In *USENIX ATC*, 2002.

[75] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1997.

[76] Open Source Software in Java. `http://java-source.net`.

[77] C. Grzegorczyk, S. Soman, C. Krintz, and R. Wolski. Isla Vista heap sizing: Using feedback to avoid paging. In *CGO*, 2007.

[78] Hadoop File System (HDFS). `http://hadoop.apache.org`.

[79] M. Hertz, Y. Feng, and E. Berger. Page-level cooperative garbage collection. Technical report, Univ. of Massachusetts, 2004.

[80] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *PLDI*, 2005.

[81] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1), 1983.

[82] A. L. Hosking. Portable, mostly-concurrent and mostly-copying garbage collection for multi-processors. In *ISMM*, 2004.

[83] A. L. Hosking and J. E. B. Moss. Protection traps and alternatives for memory management of an object-oriented language. In *SOSP*, 1993.

[84] A. L. Hosking, J. E. B. Moss, and D. Stefanović. A comparative performance evaluation of write barrier implementations. In *OOPSLA*, 1992.

[85] HotSpot Java Virtual Machine GC. `http://java.sun.com/javase/technologies/hotspot`.

[86] Hsqldb. `http://www.hsqldb.org`.

[87] R. L. Hudson and J. E. B. Moss. Incremental garbage collection for mature objects. In *IWMM*, 1992.

[88] R. L. Hudson, J. E. B. Moss, A. Diwan, and C. F. Weight. A language-independent garbage collector toolkit. Technical report, Univ. of Massachusetts, 1991.

[89] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *Operating Systems Review*, 41(2):37–49, 2007.

[90] H. Inoue, D. Stefanović, and S. Forrest. Object lifetime prediction in Java. Technical report, Univ. of New Mexico, 2003.

[91] Intel 64 and IA-32 Architectures Software Developer's Manual. Vol. 3A. System Programming Guide.

[92] Isolate API. JSR-121. `http://jcp.org`.

[93] Java 2 Enterprise Edition. `http://java.sun.com/javaee/`.

[94] JavaOS : A Standalone Java Environment, 1996. Sun Microsystems.

[95] JBoss Enterprise Middleware. `http://www.jboss.com`.

[96] JNode. `http://www.jnode.org`.

[97] R. Jones. Dynamic memory management: Challenges for today and tomorrow. In *ILC*, 2007.

[98] R. Jones and C. Ryder. Garbage collection should be lifetime aware. In *ICOOOLPS*, 2006.

[99] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

[100] H. B. M. Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9(1), 1979.

[101] H. Kermany and E. Petrank. The Compressor: Concurrent, incremental and parallel compaction. In *PLDI*, 2006.

[102] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

[103] D. Lea. A memory allocator, 1997. `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[104] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *OOPSLA*, 1998.

[105] The Linux Documentation Project. `http://tldp.org/`.

[106] M. J. M. Ma, C.-L. Wang, and F. C. M. Lau. JESSICA: Java-enabled single-system-image computing architecture. *J. Parallel Distrib. Comput.*, 60(10), 2000.

[107] J. Maassen, R. V. Nieuwpoort, R. Veldema, H. E. Bal, T. Kielmann, C. J. H. Jacobs, and R. F. H. Hofman. Efficient Java RMI for parallel programming. *Programming Languages and Systems*, 23(6), 2001.

[108] M. Macbeth, K. McGuigan, and P. Hatcher. Executing Java threads in parallel in a distributed-memory environment. In *CASCON*, 1998.

[109] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. *SIGPLAN Not.*, 40(1), 2005.

[110] S. Marion, R. Jones, and C. Ryder. Decrypting the Java gene pool. In *ISMM*, 2007.

[111] J. Mauro and R. McDougall. *Solaris Internals (2nd Edition)*. Prentice Hall, 2006.

[112] E. Meijer and J. Gough. Technical overview of the Common Language Runtime, 2000. Microsoft.

[113] Microsoft .NET Framework. `http://www.microsoft.com/net/`.

[114] D. Modberger and S. Eranian. *IA-64 Linux Kernel: Design and Implementation*. Prentice Hall, 2002.

[115] The Project Monty Virtual Machine, 2002. Sun Microsystems.

[116] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *USENIX Technical Conference*, 2005.

[117] Occam Programming Manual, 1984. Inmos Corporation.

[118] Open Source J2SE. `http://openjdk.java.net`.

[119] Oracle Database Concepts 11g Release 1, 2007. Chapter 8: Memory Architecture.

[120] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *OSDI*, 2002.

[121] Y. Ossia, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, and A. Owshanko. A parallel, incremental and concurrent GC for servers. In *PLDI*, 2002.

[122] Y. Ossia, O. Ben-Yitzhak, and M. Segal. Mostly concurrent compaction for mark-sweep GC. In *ISMM*, 2004.

[123] M. Perry. Shared Memory Under Linux, 1999. `http://fscked.org/writings/SHM/shm.html`.

[124] M. Philippsen and M. Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11), 1997.

[125] P. P. Pirinen. Barrier techniques for incremental tracing. In *ISMM*, 1998.

[126] I. Piumarta. The virtual processor: fast, architecture-neutral dynamic code generation. In *VMRTS*, 2004.

[127] T. Printezis and D. Detlefs. A generational mostly-concurrent garbage collector. In *ISMM*, 2000.

[128] T. Printezis and D. Detlefs. A generational mostly-concurrent garbage collector. *SIGPLAN Not.*, 36(1), 2001.

[129] Protocol Buffers. Google's Data Interchange Format. `http://code.google.com/p/protobuf`.

[130] R. Rashid, A. Tevanian, M. Young, et al. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *ASPLOS*, 1987.

[131] Java RMI Specification. `http://java.sun.com`.

[132] G. Rodriguez-Rivera, M. Spertus, and C. Fiterman. A non-fragmenting, non-moving garbage collector. In *ISMM*, 1998.

[133] N. Röjemo. Generational garbage collection without temporary space leaks for lazy functional languages. In *IWMM*, 1995.

[134] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *POPL*, 1988.

[135] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. *SIGPLAN Not.*, 41(10), 2006.

[136] N. Sachindran and E. Moss. MarkCopy: Fast copying GC with less space overhead. In *OOPSLA*, 2003.

[137] K. Sagonas and J. Wilhelmsson. Mark and split. In *ISMM*, 2006.

[138] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. *LNCS*, 2001.

[139] M. L. Seidl and B. Zorn. Low cost methods for predicting heap object behavior. In *WFDO*, 1999.

[140] J. Seligmann and S. Grarup. Incremental mature garbage collection using the train algorithm. In *ECOOP*, University of Aarhus, 1995.

[141] Java Object Serialization Specification. `http://java.sun.com`.

[142] A. Silberschatz. *Operating System Concepts*. John Wiley and Sons, 2004.

[143] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation, 2007.

[144] P. Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Technical report, MIT, 1988.

[145] SPEC. `http://www.spec.org`.

[146] D. Stefanovic, M. Hertz, S. M. Blackburn, K. S. McKinley, and J. E. B. Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *MSP*, 2002.

[147] W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

[148] T. Suezawa. Persistent execution state of a Java virtual machine. In *Java Grande*, 2000.

[149] A. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice Hall, 1987.

[150] A. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[151] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. *SIGOPS Oper. Syst. Rev.*, 24(3), 1990.

[152] G. Thomas, N. Geoffray, C. Clément, and B. Folliot. Designing highly flexible virtual machines: the JnJVM experience. *Softw. Pract. Exper.*, 38(15), 2008.

[153] TIOBE Index. `http://www.tiobe.com`.

[154] B. Titzer, T. Wurthinger, D. Simon, and M. Cintra. Improving Compiler-Runtime Separation with XIR. In *VEE*, 2010.

[155] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, 1984.

[156] The VolanoMark Benchmark. `http://www.volano.com/benchmarks.html`.

[157] T. von Eicken, C.-C. Chang, G. Czajkowski, C. Hawblitzel, D. Hu, and D. Spoonhower. J-Kernel: A capability-based operating system for Java. In *SIP*, 1999.

[158] IBM WebSphere Application Server. `http://www.ibm.com`.

[159] M. Wegiel and C. Krintz. The Mapping Collector: Virtual memory support for generational, parallel, and concurrent compaction. In *ASPLOS*, 2008.

[160] M. Wegiel and C. Krintz. XMem: Type-Safe, Transparent, Shared Memory for Cross-Runtime Communication and Coordination. In *PLDI*, 2008.

[161] M. Wegiel and C. Krintz. Dynamic prediction of collection yield for managed runtimes. In *ASPLOS*, 2009.

[162] M. Wegiel and C. Krintz. The single-referent collector: Optimizing compaction for the common case. *ACM Trans. Archit. Code Optim.*, 6, 2009.

[163] M. Wegiel and C. Krintz. Concurrent collection as an operating system service for cross-runtime cross-language memory management. Technical Report 15, University of California, Santa Barbara, 2010.

[164] M. Wegiel and C. Krintz. Cross-language, type-safe, and transparent object sharing for co-located managed runtimes. In *OOPSLA*, 2010.

[165] P. R. Wilson. Uniprocessor garbage collection techniques. In *IWMM*, 1992.

[166] P. R. Wilson. Uniprocessor garbage collection techniques. Technical report, Univ. of Texas, 1994.

[167] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *IWMM*, 1995.

[168] P. R. Wilson, M. S. Lam, and T. G. Moher. Caching considerations for generational garbage collection. In *LFP*, 1992.

[169] P. R. Wilson and T. G. Moher. A card-marking scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87–92, 1989.

[170] N. Wirth and J. Gutknecht. *Project Oberon: the design of an operating system and compiler*. ACM Press/Addison-Wesley, 1992.

[171] D. S. Wise. Stop-and-copy and one-bit reference counting. Technical report, Indiana University, 1993.

[172] D. S. Wise and D. P. Friedman. The one-bit reference count. *BIT*, 17(3):351–9, 1977.

[173] F. Xian, W. Srisa-an, and H. Jiang. MicroPhase: An approach to proactively invoking garbage collection for improved performance. In *OOPSLA*, 2007.

[174] T. Yang, E. D. Berger, M. Hertz, S. F. Kaplan, and J. E. B. Moss. Autonomic heap sizing: Taking real memory into account. In *ISMM*, 2004.

[175] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *OSDI*, 2006.

[176] Z. Yang and K. Duddy. Corba: A platform for distributed object computing (a state-of-the-art report on omg/corba). *Operating Systems Review*, 30:4–31, 1996.

[177] L. Youseff and R. Wolski. Vshmem: Shared-memory os-support for multicore-based hpc systems. Technical report, University of California, Santa Barbara, 2009.

[178] W. Yu and A. L. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11), 1997.

[179] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara. Program-level adaptive memory management. In *ISMM*, 2006.

[180] B. Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *LFP*, 1990.

[181] B. Zorn. The measured cost of conservative garbage collection. *Software Practice and Experience*, 23:733–756, 1993.

[182] B. Zorn and M. Seidl. Segregating heap objects by reference behavior and lifetime. In *ASPLOS*, 1998.